

Programming with DirectToSOM™ C++

- *Covers SOM, DSOM, and CORBA Object Services*
- *Includes Release-to-Release Binary Compatibility (RRBC) Support*
- *Examples of Interlanguage Object Sharing*

JENNIFER HAMILTON



Includes
Disk

Programming with DirectToSOMTM C++

JENNIFER HAMILTON

Wiley Computer Publishing



John Wiley & Sons, Inc.

New York • Chichester • Brisbane • Toronto • Singapore • Weinheim

Publisher: Katherine Schowalter
Editor: Theresa Hudson
Managing Editor: Micheline Frederick
Electronic Products, Associate Editor: Mike Green
Text Design & Composition: North Market Street Graphics

Designations used by companies to distinguish their products are often claimed as trademarks. In all instances where John Wiley & Sons, Inc., is aware of a claim, the product names appear in initial capital or ALL CAPITAL LETTERS. Readers, however, should contact the appropriate companies for more complete information regarding trademarks and registration.

This text is printed on acid-free paper.

Copyright © 1997 by John Wiley & Sons, Inc.
All rights reserved. Published simultaneously in Canada.

This publication is designed to provide accurate and authoritative information in regard to the subject matter covered. It is sold with the understanding that the publisher is not engaged in rendering legal, accounting, or other professional service. If legal advice or other expert assistance is required, the services of a competent professional person should be sought.

The opinions expressed in this book are those of the author and do not represent those of her employer.

Reproduction or translation of any part of this work beyond that permitted by section 107 or 108 of the 1976 United States Copyright Act without the permission of the copyright owner is unlawful. Requests for permission or further information should be addressed to the Permissions Department, John Wiley & Sons, Inc.

Library of Congress Cataloging-in-Publication Data:

Hamilton, Jennifer, 1965–

Programming with DirectToSOM C++ / Jennifer Hamilton.

p. cm.

Includes index.

ISBN 0-471-16004-0 (pbk. : acid-free paper)

1. Object-oriented programming (Computer science) 2. C++
(Computer program language) I. Title.

QA76.64.H356 1996

005.13'3—dc20

96-34366

CIP

Printed in the United States of America
10 9 8 7 6 5 4 3 2 1

*To my parents, Mary and Stan, who taught me to work hard
and enjoy learning.*

About the Author

Jennifer Hamilton holds a BSc Honours in Computer Science from the University of Victoria, BC, and is currently completing a Computer Science MMath with the University of Waterloo. She has worked in compiler development since joining the IBM Toronto Laboratory in 1987. Jennifer is the author of two books and numerous articles on programming language-related topics, and is currently responsible for developing the DirectToSOM C++ support for IBM's future-generation C++ language product.

Acknowledgments

Many people have contributed to this book, both directly and indirectly. Mark Mendell and Brian Thomson endured a million questions, and were great to work with, as I struggled to learn about SOM, DirectToSOM C++, and our VisualAge C++ compiler all at once. Liane Acker was extremely helpful in answering all my questions about DSOM. The DSOM chapter would not be what it is without her advice and technical expertise. Jacob Slonim encouraged me to research and write about SOM, which led in part to this book. John Ferreira got me started with DirectToSOM C++ development and encouraged me to write about it.

I would also like to thank and acknowledge the many people contributed to this book by reviewing the material and providing valuable comments and suggestions; in particular, Mark Mendell. Also Liane Acker, Ray Barsness, Scott Danforth, John Ferreira, Robert Klarer, John O'Keefe, Catherine Lung, Stephen Miller, Ashvin Radiya, Judd Rogers, Pete Schommer, Roger Sessions, Kevin Sutter, Nick Tindall, Brian Thomson, and Wai Ming Wong.

Portions of the material in this book originally appeared in the *C++ Report* magazine and the proceedings of the *2nd USENIX Conference on Object-Oriented Technologies and Systems*, and has been reproduced with the permission of the respective publishers.

F O R E W O R D

Object-oriented programmers dream of a world with vast collections of reusable components, transforming our industry from its current “build from scratch” development paradigm to one where programmers simply “assemble” components into applications. We expect that the ability to build on the work of others will tremendously improve our ability to construct new applications that have far greater functionality and are of much higher quality than those we could have built from scratch. However, despite the great strides we have made in object technology, especially in languages and tools, we are still limited by our technology for packaging objects into binary libraries that can be used across applications and across programming languages. Microsoft has developed a limited solution to the problem called component object model (COM) which has become the cornerstone of its current software strategy. But, even before COM was developed, IBM, with the backing of its more visionary executives (especially Vice President of Software Technical Strategy Larry Loucks), began work on a much more ambitious solution to the object packaging problem, the System Object Model (SOM). SOM first shipped in OS/2 2.0 in March 1992.

SOM, like COM, provides a way to develop libraries of reusable binary object implementations. Unlike COM, SOM fully supports object-oriented technology. SOM provides a common object run time that can be tightly integrated into many programming languages and development tools and that allows objects to be shared across different languages or different implementations of the same language. SOM also supports release-to-release binary compatibility, or RRBC. RRBC means that an object library provider can evolve his or her library in almost any way that would not require a using-application's source to be recoded without requiring the application to be recompiled. For example, one can change the size of objects, add additional methods, change the implementation of methods, or even add new object classes into a class hierarchy without breaking binary compatibility with existing compiled applications that dynamically link to the object's implementation.

SOM is a key component in IBM, Apple, and other companies' software strategies. It is available now (or will be very soon) on almost all IBM systems, the Apple Macintosh, Microsoft Windows 95, Microsoft Windows NT, and Tandem's Guardian system. SOM is also the object foundation for the industry-wide effort of OpenDoc. For several years, SOM has supported the Object Management Group's CORBA standard for distributed computing, which allow methods to be called on objects wherever the objects are located. In addi-

tion, IBM has recently developed a way for SOM objects to appear as COM and/or OLE objects. Now, developers using COM/OLE-based tools (such as Visual Basic) can directly leverage libraries of SOM objects without even being aware that SOM is involved. This capability includes the ability to access remote SOM objects. This support is expected to ship later this year for Windows 3.1, Windows 95, and Windows NT.

SOM support exists for a number of programming languages including C, C++, Smalltalk, and COBOL, with Java support coming soon. You can get late-breaking news about SOM, interact with other SOM users, and get free downloads of SOM technology from the SOM Web page at: <http://www.software.ibm.com/objects/somobjects>.

This book covers one of the most exciting recent developments in SOM technology—the ability to directly and seamlessly leverage SOM within a C++ development environment. This technology is called DirectToSOM (DTS) C++. It allows a programmer to “just program in C++” and yet implement, use, and subclass SOM-based objects. DTS C++ is the result of major effort between IBM and the Metaware Corporation to develop a complete mapping of the full set of C++ language features to the SOM run time. This book is a thorough introduction to this exciting technology, providing numerous practical examples of its use. With the delivery of the IBM VisualAge DTS C++ development environment, C++ developers can leverage their C++ skills in a highly productive integrated development environment to produce robust, reusable, and language-neutral binary libraries of object definitions. DTS C++ allows you to fully exploit the capabilities of SOM and C++. Even if you are concerned only with in-house C++ development, you will find that the release-to-release binary compatibility you get with DTS C++ gives an enormous productivity boost to team development because it greatly reduces the need for massive rebuilds of your software every time one of your base classes changes. IBM Visual Age C++ with DTS support is available now for Windows NT, Windows 95, OS/2, AIX, and MVS, with more platforms coming soon.

Jennifer Hamilton is a member of the VisualAge C++ compiler team responsible for developing the compiler's DTS support. She has drawn upon her unique experience and her access to the rest of the technical team to bring you this book. It contains all of the insight and practical information you need to get started immediately enjoying the benefits of a state-of-the-art C++ development environment combined with all the advanced object technology of SOM. It covers the SOM technology and shows you how to take full advantage of SOM, including RRBC, distribution, and cross-language support. Extensive use of code examples will help you get started quickly and the sections on distributed SOM will give you all the practical advice you need to leverage the exciting world of CORBA-based distributed computing.

Enjoy.

Mike Conner, Ph.D.

Mike Conner is a Senior Member of the Technical Staff at IBM and is the creator of SOM. He can be reached at mikec@austin.ibm.com.

C O N T E N T S

Preface	ix
List of Figures	xii
Chapter 1: Introduction to SOM	1
The Disadvantages of Object-Oriented Programming Languages	1
The System Object Model	4
DirectToSOM C++	10
Summary	12
Chapter 2: DirectToSOM C++ Overview	15
Defining DirectToSOM C++ Classes	15
Basic Concepts	18
Setup	24
Chapter 3: Release-to-Release Binary Compatibility	35
Supporting RRBC	35
Supported Changes	39
Unsupported Changes	43
A Complete Example	44
RRBC Usage Considerations	50
Chapter 4: Using DirectToSOM C++	51
DirectToSOM C++ Pragma's	51
Macros Defined for DirectToSOM	80
Compiler Options	80
Chapter 5: Programming Considerations	83
Differences between Native C++ and DirectToSOM C++	83
Programming Considerations	92
Common Programming Problems	124
Chapter 6: Inside DirectToSOM C++	133
The SOM Object Model	133
SOM Class Data Structures	140
SOMObject Methods	148
Name Mangling	151

Chapter 7: IDL Generation	155
Generating an IDL File	155
Mapping DirectToSOM C++ Classes to IDL	162
Mapping Types from C++ to IDL	174
Other Mappings	181
Generating IDL-Specific Information from C++	190
Generating DirectToSOM Class Definitions from IDL Definitions	194
The Interface Repository	201
Chapter 8: Distributed SOM	203
DSOM Overview	203
DSOM Programming Considerations	217
Putting It All Together	252
Common DSOM Problems	264
DSOM 2.x	269
Chapter 9: Interlanguage Object Sharing	275
Introduction	275
Smalltalk	276
OO COBOL	278
Style Guidelines for Defining DTS C++ Classes	282
Examples	284
Chapter 10: The SOMObjects Object Services	305
Overview	305
Naming Service	312
Life Cycle Services	324
Object Identity Service	329
Externalization Service	332
Persistence	341
Appendix A: SOMObject Header Files	377
<somapi.h>	377
<somcorba.h>	393
<som.hh>	398
<somh.hh>	398
<somobj.hh>	399
<somcls.hh>	404
Appendix B: Persistence SOM	415
Persistent Objects: An Overview	415
The Persistent ID	419
Restoring Persistent Objects	423
Working with Persistent Objects	426
Persistent Message Queue	429
Programming Considerations and Common Problems	439
Migration Considerations for POSSOM	440
References	445
Index	447

IBM's System Object Model (SOM) is the underpinning of IBM's object-oriented strategy. SOM was first made available with version 2.0 of OS/2 in 1992, and has since been made available on AIX, MVS, Windows 3.1, Windows 95, and Windows NT. SOM conforms with the industry standard Common Object Request Broker Architecture (CORBA), produced by the Object Management Group (OMG). In fact, SOM was the first product to implement CORBA-compliant distributed object support. Most recently, SOMObjects version 3.0 is currently available in beta for OS/2. This is significant in that it supports the CORBA object services that have been standardized by the OMG.

Most of the existing literature on SOM focuses on using the CORBA Interface Definition Language (IDL) to define a class, from which C or C++ program skeletons (called *language bindings*) are generated to implement the class. While this approach allows SOM classes to be implemented using any compiler, as no special language support is required, it may not be very appealing to C++ programmers for several reasons. One drawback is that classes must be written in two different languages, IDL and C++. Generating program skeletons is a bit unnatural, in that the program style and organization is never what you would get if you wrote it initially yourself, as everyone has his or her own programming style. Further, using the language bindings restricts you to a subset of the C++ language, both restricting the ability to take advantage of language features and making it very difficult to port existing class libraries to SOM.

DirectToSOM C++ allows programmers to take advantage of the power of SOM directly in the C++ programming language, without having to first write in IDL and then generate C++. Through the underlying SOM support, DirectToSOM C++ brings powerful release-to-release binary compatibility (RRBC) features to the C++ language, allowing classes to be updated without requiring recompilation of client code, which is a problem for most C++ implementations today. RRBC is an important feature, lacking in most C++ compilers, that is starting to receive considerable focus in the industry. In addition, DirectToSOM C++ provides the capability to distribute objects through DSOM and share objects with other languages such as Smalltalk, all using standard C++ syntax with few language restrictions.

DirectToSOM C++ is a cross-platform product that is part of the IBM VisualAge C++ and Metaware High C++ compiler products. It is currently available for OS/2, Windows 95, Windows NT, AIX, and MVS. The focus of this book is to explain how to use SOM from a DirectToSOM C++ perspective. The reader is expected to be familiar with the C++ language, but not with SOM.

I wrote this book to address the need for detailed information, and in particular, programming examples covering DirectToSOM C++. My goal was not to explain every last detail of SOM, but to explain the basics, in particular anything that is pertinent to DirectToSOM C++. As with any product, having working examples to follow when you are getting started can help tremendously. I have provided many complete programming examples, even for relatively simple concepts, and particularly for more complicated areas such as DSOM. In addition, I cover many of the common programming paradigms and problems that are encountered, based on my experience in the development of the DirectToSOM C++ support for IBM's VisualAge C++ products.

How This Book Is Organized

The book is organized into the following chapters:

Chapter 1, *Introduction to SOM*, explains the basics of SOM. Chapter 2, *DirectToSOM C++ Overview*, introduces DirectToSOM C++ and explains basic concepts. Chapter 3, *Release-to-Release Binary Compatibility (RRBC)*, discusses the capabilities and limitations of the DirectToSOM C++ RRBC support. Chapter 4, *Using DirectToSOM C++*, describes the DirectToSOM C++ pragmas and compiler options. Chapter 5, *Programming Considerations*, explains various programming paradigms, restrictions, and semantic differences between native C++ and DirectToSOM C++. Chapter 6, *Inside DirectToSOM C++*, describes how the C++ object model is mapped to SOM. Chapter 7, *IDL Generation*, explains how C++ is mapped to IDL, along with any restrictions or programming considerations. Chapter 8, *Distributed SOM*, explains how to distribute objects using DSOM and DirectToSOM C++, including data access and memory management considerations. Chapter 9, *Interlanguage Object Sharing*, shows how to share objects between DirectToSOM C++, IBM Smalltalk and OO COBOL. Chapter 10, *The SOMObjects Object Services*, provides an overview of the Object Services along with examples of using several with DirectToSOM C++. Appendix A, *SOMObject Header Files*, contains listings of several of the SOMObjects header files. Appendix B, *Persistence SOM (PSOM)*, describes the original SOM support for persistence, which will be replaced by the Persistent Object Services (POSSOM).

About the Diskette

The diskette accompanying this book contains the source code for all the examples in the book. The diskette is organized into directories corresponding to the book chapters. Individual examples are identified in the text by their file and directory name within each chapter directory.

The source code has been tested on OS/2 with the SOMObject 3.0 beta and VisualAge C++ for OS/2 Version 3.0 with CTC304. The examples up to chapter 8 have been tested on Windows NT with SOMObjects 2.1 and VisualAge C++ for Windows Version 3.0, both at the GA level.

For the DSOM 2.1 and PSOM examples, SOM header files are also included which fix problems in the shipped and generated header files for OS/2 (these fixes are available through the latest CSD for SOMObjects 2.1 for OS/2).

Conventions Used in This Book

Throughout the text, several fonts are used to identify various programming elements. Boldface is used to identify those symbols that are defined by the operating system, SOM, or the various compiler products. For example, **SOMObject** is the name of a SOM-defined class, so it will appear in bold when referred to in the text. Monospace font is used to refer to symbols that are user-defined as part of the programming examples. For example, the function `checkError` is a user-defined function that is not supplied by any software product. All programming examples and program excerpts appear in monospace.

List of Figures

Figure 1.1 Updating a C++ Class Definition	2
Figure 1.2 No Standard Object Representation	3
Figure 1.3 Updating a SOM Class Definition	5
Figure 1.4 SOM Defines a Standard Object Representation	6
Figure 1.5 SOM Usage Model	12
Figure 2.1 SOM Class Object	23
Figure 2.2 Message Queue Data Structure Diagram	27
Figure 5.1 Implementation of DirectToSOM C++ Object	89
Figure 5.2 Class with Embedded DirectToSOM C++ Objects	89
Figure 6.1 SOM Object Run-time Model for Class Hello	134
Figure 6.2 SOM Object Structure	136
Figure 6.3 Creating Object through somNew	137
Figure 6.4 Creating Class Object through NewClass	140
Figure 6.5 Creating Class Object with somBuildClass	141
Figure 6.6 SOM Class Data Structures	145
Figure 8.1 DSOM Overview	204
Figure 8.2 DSOM Server Process	211
Figure 8.3 Process Layout With Factory	213
Figure 8.4 Creating Objects of Same Type in Different Servers	219
Figure 8.5 Caller-owned Memory Management	222
Figure 8.6 Object-Owned Memory Management	224
Figure 8.7 Dual-owned Memory Management	226
Figure 8.8 Server Process Model	251
Figure 8.9 DSOM 2.x Object/Process Layout with Server Object	273
Figure 9.1 Language Access to SOM Class Descriptions	276
Figure 10.1 Managed Object Life Cycle	308
Figure 10.2 Object Services Server	311
Figure 10.3 Stream Object Representation	333
Figure 10.4 Saving a POSSOM Object	346

Introduction to SOM

The Disadvantages of Object-Oriented Programming Languages

A major goal of object-oriented programming is to write programs that can be more easily reused and extended than those written using conventional programming practices. While there is considerable debate in the industry as to just how successful object-oriented programming languages have been with respect to achieving greater software reuse, such discussions typically apply to the source code only. But what about binary reuse? This is an issue that is not solved by languages such as C++, and is actually made considerably more difficult. A major advantage of object-oriented programming languages over procedural programming languages is support for encapsulation, which groups data with associated methods. However, this grouping also introduces binary reuse problems, specifically in the area of *release-to-release binary compatibility* (RRBC), which is the ability to update a program binary without impacting existing clients, and interlanguage object sharing. C++, arguably the most commonly used object-oriented programming language, suffers in particular from these problems. Binary reuse is actually more difficult to achieve using C++ than using its predecessor, the procedural C programming language.

When using C, new versions of library routines can be introduced without impacting existing code, provided that the procedure signatures are kept compatible and new function names don't collide with existing client names. While keeping signatures compatible and avoiding name collisions can sometimes be difficult, it is a relatively simple problem compared to that of keeping class definitions compatible in C++. The problem is that a large amount of information about a C++ class, such as its instance size, the order and location of methods, and the offset to parent class data, is compiled into client code. Thus, adding a new data member to a class, even a completely private member, in most cases requires recompilation of client code, including subclasses. Figure 1.1 illustrates what can happen if a C++ class is updated, and the client is not recompiled. In the first panel, both the implementation and the client have been compiled with the same version of class A, which becomes part of the binary for those two programs. In the second panel, the class definition has been updated to A', and only the implementation has been recompiled. This results in an incompatibility between the client's view of the class and the implementation. The client must also be recompiled, as shown in the final panel, in order to manipulate objects that are compatible with the implementation. In some cases, binary compatibility can be achieved by carefully managing class changes, but migrating a method up the class hierarchy or inserting a new class in the hierarchy always requires recompilation of client code.

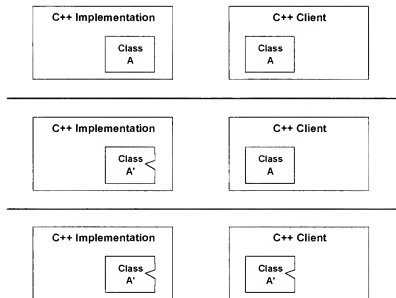


FIGURE 1.1 Updating a C++ class definition.

C++ also impedes the sharing of code between languages, as shown in Figure 1.2. It is relatively easy to call a C library routine from Fortran, or vice-versa, but very difficult, if not impossible, to share objects between C++ and languages such as Smalltalk. This is because each language introduces a specific, internal structure for representing object data and associated methods. There is no standard object representation, such as operating system linkage conventions for procedural languages, to enable the sharing of objects across different languages. Even sharing objects across different C++ implementations is not readily achievable. There is no standard object representation defined for C++ objects, so each compiler vendor must choose one. Most use a variant of the virtual function table scheme suggested in the *Annotated C++ Reference Manual* (Ellis, 1990), but there is no requirement that the virtual function table scheme be used at all. However, unless the layout is identical between two compiler vendors, objects cannot be shared between these implementations.

As a result, commercial C++ class library vendors cannot ship a single binary to their customers with the expectation that it will be useful regardless of which compiler or programming language their customers use. Class library and framework vendors must ship either source code or a separate precompiled version of their products for each different compiler that their customers are using. Clearly, we have a problem: there is no guaranteed way to share objects across different C++ implementations, let alone with other object-oriented programming languages such as Smalltalk, and limited ability to make common updates to classes without requiring recompilation

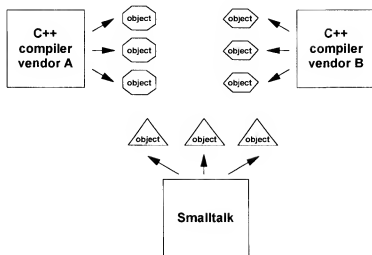


FIGURE 1.2 No standard object representation.

of client code. Object-oriented programming is intended to promote code reuse and allow changes to be made to class implementations without affecting client code. But this source-level solution introduces a new set of problems at the binary object level: release-to-release binary compatibility and interlanguage object-sharing.

As there is much work being done in the area of C++ class libraries and frameworks, it is particularly important to solve the binary object problem so that class library providers can supply updated versions of their classes without impacting existing code. Further, class libraries must be usable from different languages, or at the very least different language implementations, without requiring multiple versions of the library for each target language or implementation.

The System Object Model

The *System Object Model* (SOM) was designed to provide a state-of-the-art object model that addresses the two problems introduced by object-oriented programming languages: release-to-release binary compatibility and interlanguage object sharing. SOM provides separation of interface and implementation through a language-independent object model, allowing the class client and implementation to be written in different languages. SOM is not a programming language, nor does it support one. Rather, it is intended to augment the support provided in existing programming languages. The real contribution of SOM is to provide a means of describing, packaging, and using objects through various languages so that binary compatibility and language independence can be achieved.

SOM allows a new version of a class to be supplied without requiring recompilation of any unmodified client code. In general, making a change to a SOM class that does not require a source code change in a client, such as adding new methods, instance variables, or even additional base classes, does not require recompilation of that client. SOM also includes Distributed SOM (DSOM), allowing objects to be shared between processes, or even across networks, and is fully compliant with the Object Management Group's (OMG) Common Object Request Broker Architecture (CORBA) standards.

SOM supports a model similar to that of Smalltalk in that classes are not purely syntactic entities, as in C++, but are themselves objects. SOM class objects are created at run time as required by the client, and are used for creating and manipulating instances. Class objects support a variety of methods for creating and querying objects, such as determining the size of class instances, whether a method is supported by a given class or a given object is a member of that class.

With the C++ model, as shown in Figure 1.1, all class information is statically compiled into the program binary, impeding RRBC. With the SOM

model, much of the class information is kept separate from the program binary and is accessed at run time through the SOM class object, as shown in panel 1 of Figure 1.3. When the implementation is recompiled with an updated version of the class, panel 2, the SOM class object will support that new implementation. However, because the client does not have static dependencies on the class structure, the updated implementation can be binary-compatible to the previous version, supporting the earlier view of the class expected by the existing client. This model allows changes to be made to the class implementation, such as changing the size of instances, in a way that will not require recompilation of the client program.

As with most object-oriented systems, SOM objects are run-time entities that support a specific interface and have an associated state and implementation. This implementation is not accessible except through the SOM object. The SOM run time controls the layout and direct manipulation of class instances. All manipulation of SOM objects is performed through standard procedure calls to the SOM API. By restricting the layout and access to objects, SOM defines and enforces a standard object model that allows objects to be shared across C++ implementations and between different programming languages, as shown in Figure 1.4.

One of the ways SOM supports RRBC is by maintaining a list of all methods introduced by a class, called the *release order* for the class. The release order is used by clients to access methods in the method table for the class, and is the only dependency that a client has upon a corresponding class implementation. By keeping the order of methods in this list consistent, new methods can be added to a SOM class without forcing recompilation of client code. For C++, SOM also keeps track of the instance data for a

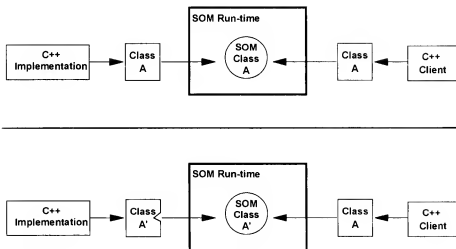


FIGURE 1.3 Updating a SOM class definition.

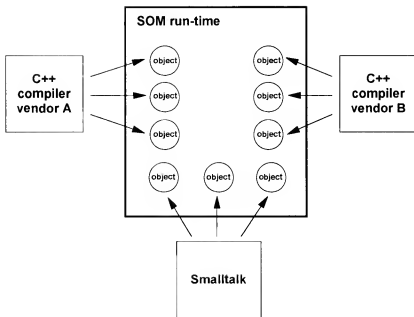


FIGURE 1.4 SOM defines a standard object representation.

class, and provides addressability to that data upon request through the class object at run time. This model both supports RRBC for C++ classes and allows classes to be shared between different C++ implementations.

SOM Components

The SOMObjects Developer Toolkit, which is used for developing SOM applications, consists of several components: the SOM compiler, the SOM run-time library, SOM frameworks and class libraries, and the Object Services.

The SOM compiler interprets SOM class definitions, written in the CORBA standard language called the Interface Definition Language (IDL). IDL is a language-neutral means for describing object interfaces, allowing different compilers and even different programming languages, to manipulate shared objects. The IDL definition describes the interface to, but not the implementation of, a SOM class. Among other things, the SOM compiler generates *bindings* for a given target language from the IDL description. Bindings are language-specific macros and procedures that allow a programmer to interact with SOM through a simplified syntax that is more natural for the particular language.

The SOM run-time library supports the run-time creation and use of SOM objects. It provides a variety of classes, methods, and procedures that

support the interaction between class clients and implementations. The library can be used to manipulate SOM objects and classes through any language that supports external procedures and procedure pointers, and that can map the IDL types to its own type system.

There are several frameworks and class libraries provided with the SOMObjects Toolkit, including:

- ♦ *Distributed SOM (DSOM)*: A framework that allows SOM objects to be created and shared across a network.
- ♦ *Interface Repository Framework*: A CORBA-compliant framework that supports access to the interface repository, which is a database, optionally created by the SOM compiler, that contains information about a SOM class.
- ♦ *Emitter Framework*: A framework that allows customized information to be created by the SOM compiler from an IDL specification. For example, the language bindings are created through SOM-supplied emitters.
- ♦ *Metaclass Framework*: SOM class objects are instances of metaclasses. The metaclass framework allows you to customize the semantics of object instantiation and method invocation.
- ♦ *Event Management Framework*: A framework that allows a single-threaded application to respond to multiple events in a single main loop.
- ♦ *Collection classes*: Classes that provide a variety of commonly used container interfaces.

The Object Services are new for SOMObjects 3.0. They implement a variety of the CORBA Common Object Services Specifications (OMG Document Number 95-3-31), including Concurrency, Events, Externalization, Naming, LifeCycle, Object Identity, Persistence, and Transactions.

Earlier versions of the SOMObjects Toolkit, prior to version 3.0, also included two other frameworks:

- ♦ *Persistence SOM*: A framework that allows objects to be saved in a persistent data store so that they may exist beyond the lifetime of the program in which they were created. This framework has been replaced by the CORBA Persistence Object Service in SOMObjects 3.0.
- ♦ *Replication SOM*: A framework that allows objects to be replicated and simultaneously updated across a network. This framework has been discontinued, without replacement, in SOMObjects 3.0.

Defining SOM Classes

Initially, programmers could define SOM classes only through IDL, using the SOM compiler to generate language bindings for a given programming language. Currently, the SOM compiler generates bindings for the C and

C++ languages. The C++ bindings, for example, allow SOM objects to be manipulated through C++ pointers to objects, using any C++ compiler. Again, SOM does not support a programming language, only an implementation definition language. Other than creating the bindings files, which facilitate access to the SOM API, SOM does not provide any mechanism for creating class implementation or clients.

The following program example shows the IDL definition for the class SOM class `Hello`, with the single method `sayHello`.

```

1 #include <somobj.idl>
2
3 interface Hello : SOMObject
4 {
5     void sayHello();
6 };

```

The SOM compiler will generate usage bindings, implementation bindings, and an implementation template for the class. The *usage bindings* define the public interface to a SOM class, and are included by clients to create and manipulate objects of that class. The *implementation bindings*, included by the class implementor, define the class and include private information that is not part of the class interface. Both the implementation and usage bindings files are regenerated completely by the SOM compiler when the class is modified, and should not be updated directly. (They are also quite lengthy, so they will not be shown for the sake of brevity.) The *implementation template* contains procedure stubs for each method introduced or overridden by the class, and is updated incrementally by the SOM compiler when methods are added or a method signature is modified. The class implementor modifies the implementation template file to define the class behavior.

As an example of the C++ bindings, the following shows the generated C++ implementation template for the SOM class `Hello` from the preceding IDL definition. Assuming that the IDL definition is contained in the file `hello.idl`, the SOM compiler will generate the usage and implementation bindings and the implementation template for the class `Hello` in the files `hello.xh`, `hello.xih`, and `hello.cpp` respectively. The implementation binding file for the class `Hello` is included at line 12 from the file `hello.xih`.

The implementation template for the method `sayHello` is shown in lines 14 through 18. The **SOM_Scope** and **SOMLINK** macros provide system-specific information for SOM. The first parameter to a SOM method is the target SOM object, from which the address of the instance data can be retrieved by calling the function `classNameGetData`, shown commented out at line 16. The second parameter is an environment parameter that is required for CORBA compliance. `HelloMethodDebug` is a macro that optionally displays debugging information.

```

1  /*
2  * This file was generated by the SOM Compiler and Emitter Framework.
3  * Generated using:
4  *   SOM Emitter emitxtm.dll: 2.41
5  */
6
7  #ifndef SOM_Module_hello_Source
8  #define SOM_Module_hello_Source
9  #endif
10 #define Hello_Class_Source
11
12 #include "hello.xih"
13
14 SOM_Scope void SOMLINK sayHello(Hello *somSelf, Environment *ev)
15 {
16     /* HelloData *somThis = HelloGetData(somSelf); */
17    >HelloMethodDebug("Hello", "sayHello");
18 }

```

Once the bindings and template files have been generated, the next step is to fill in the class implementation. In this example, we want the method `sayHello` to print the message “Hello world” (note that I have deleted some of the generated lines to simplify the example):

```

1  #include <iostream.h>
2  #include "hello.xih"
3
4  SOM_Scope void SOMLINK sayHello(Hello *somSelf, Environment *ev)
5  {
6    >HelloData *somThis = HelloGetData(somSelf); */
7    >HelloMethodDebug("Hello", "sayHello");
8
9     cout << "Hello world" << endl;
10 }

```

Now that we have completed the class implementation, we will create a client program that uses the class `Hello`, shown next. With the C++ bindings, SOM objects are declared and manipulated as pointers to the given class. The **new** operator is used to create class instances. The first time an object of any class is created, the SOM run-time environment will be initialized implicitly. And the first time an instance of a given class is created, the associated class object will be created, along with any parent class objects.

In the following code example then, line 7 in the function `main` will initialize the SOM run-time environment, create a class object `Hello`, and allocate storage for an instance of the class `Hello`, which will be assigned to the variable `obj`. Line 8 invokes the method `sayHello` on the object `obj`, while line 9 deallocates the storage for the object. This program and the class client can be compiled and run with any C++ compiler. In addition, differ-

ent C++ compilers can be used to compile the client program and the class implementation: SOM's language-neutral object model enables this separation of interface and implementation. There is much more to the C++ bindings than is shown here, but you should have a feel for how SOM classes are defined, implemented, and used through the C++ bindings.

Note that the implementation for the method `sayHello` is a simple function, whereas the method is invoked using a standard method call. As part of the separation of interface and implementation, the usage bindings convert the client method invocation to the target implementation code through the SOM API. This conversion will be discussed in more detail in Chapter 6 *Inside DirectToSOM C++*.

```

1  #include "hello.xh"
2
3  int main(int argc, char *argv[])
4  {
5      Hello *obj;
6
7      obj = new Hello;
8      obj->sayHello(somGetGlobalEnvironment());
9      delete obj;
10     return(0);
11 }
```

DirectToSOM C++

The capability to generate C++ bindings from an IDL description enables you to create and manipulate SOM objects with any, or multiple, C++ compilers, gaining the advantages of the binary compatibility support provided by SOM. In addition, those objects can be shared across different C++ implementations or even with different languages such as Smalltalk. However, in using the C++ bindings, you are limited to a subset of the C++ language, making migration of existing C++ applications more difficult, and you must use two languages (IDL and C++) to define and manipulate objects. With the advent of DirectToSOM C++ compilers, C++ programmers can now define SOM classes through the C++ programming language directly.

DirectToSOM (DTS) C++ compilers support and enforce both the C++ and the SOM object models, allowing C++ programmers to take advantage of SOM through C++ language syntax and semantics so that the use of SOM is reasonably transparent and efficient. Instead of first describing SOM classes in IDL, the DirectToSOM C++ compiler translates C++ syntax to SOM. You can then have the compiler generate IDL from your C++ declaration, or you may find that you don't need to deal with IDL at all and can work exclusively in DirectToSOM C++. And, because you write C++ directly, you can use C++ features in your SOM classes that are available only with DirectToSOM C++,

features including templates, operators, constructors with parameters, default parameters, static members, public instance data, and more.

A major inhibitor to RRBC with C++ is the fact that so much information about an object is statically compiled into client code, in particular the location of instance data and virtual function pointers. Data layout and method calling for a DirectToSOM C++ class are done using the SOM API, instead of the native C++ API. When you run a program defining a DirectToSOM C++ class, the compiler will create the corresponding SOM class object at run time and use it to create and manipulate the object. As a result, unlike a standard C++ object, much of the information about a SOM object and its class, such as the instance size, is not determined until run time when the class object is created. This enables class evolution without forcing recompilation of clients applications.

A C++ class is made into a DirectToSOM C++ class by inheriting from the class **SOMObject**, which is defined in the header file **<som.hh>**. You can do this explicitly, as shown in the next example, or implicitly through compiler switches or pragmas that insert **SOMObject** as a base class. The access specifiers **private**, **protected**, and **public** are supported for SOM classes and enforced following the C++ rules, as are constructors and destructors and most other C++ constructs.

The following example shows a complete DirectToSOM C++ class description (**hello.hh**), implementation (**hello.cpp**), and client (**main.cpp**) corresponding to the IDL and C++ bindings example shown earlier. However, no IDL is necessary with DirectToSOM C++—the compiler implicitly maps the class to the SOM object model. As with the language bindings, the first time an object of any class is created, the SOM run-time environment will implicitly be initialized (in this example, at line 5 of the function **main** in **main.cpp**). And the first time an instance of a given class is created, the associated class object will be created, along with any parent class objects.

DirectToSOM C++ description of class Hello (hello.hh):

```
1 #include <som.hh>
2
3 class Hello : public SOMObject {
4     public:
5         void sayHello();
6 };
```

DirectToSOM C++ implementation of class Hello (hello.cpp):

```
1 #include <iostream.h>
2 #include "hello.hh"
3
4 void Hello::sayHello()
5 {
```

Continued

```

6     cout << "Hello world" << endl;
7 }

```

DirectToSOM C++ client of class Hello (main.cpp):

```

1 #include "hello.hh"
2
3 int main()
4 {
5     Hello obj;
6
7     obj.sayHello();
8 }

```

Summary

Figure 1.5 shows the various ways that a SOM class can be defined and used. The top panel shows the different mechanisms for describing SOM

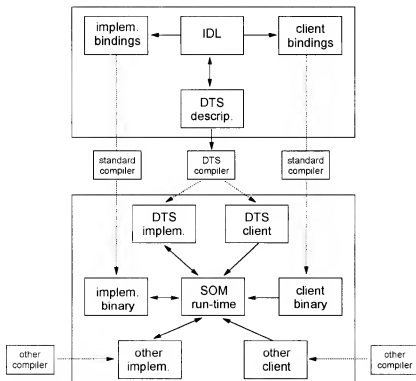


FIGURE 1.5 SOM usage model.

classes. An IDL description can be used by the SOM compiler to generate client and implementation bindings in supported languages, which are then used as input to a standard compiler to create client and implementation binaries. The SOM compiler can also generate DirectToSOM (DTS) C++ class definitions, which are used as input to a DirectToSOM C++ compiler to create client and implementation bindings. In addition, DirectToSOM compilers can produce IDL definitions from DirectToSOM class descriptions. Regardless of the mechanism used to define the class, as shown in the bottom panel, all SOM clients and implementations interact through the SOM run time. Note that a class client or implementation could be written using a language for which no language bindings or DirectToSOM support is available ('other' in the figure). SOM objects can be accessed from any language that supports external procedure calls and procedure pointers and that can map IDL types onto the native language types. Specialized emitters can be written using the emitter framework to make this easier.

The C++ language suffers from an inability to support class library evolution or interlanguage object sharing. SOM provides a language-neutral object model with support for RRBC and distributed objects, and that is CORBA-compliant. You can implement, create, and use SOM objects from almost any language, but language bindings make that process much easier. For C++ programmers, most SOM development was initially done using the C++ bindings, which allow SOM objects to be manipulated without special compiler support. The disadvantages of this approach are restrictions in language support and the necessity to use two programming languages. The advent of DirectToSOM C++ compilers gives C++ programmers the power of SOM through programming directly in C++, thereby providing RRBC and interlanguage object-sharing support for the C++ language.

Programming with DirectToSOM C++ describes how to develop applications using DirectToSOM C++, including the various pragmas and compiler options available, and DirectToSOM C++ programming considerations, and an explanation of how the C++ object model is mapped to the SOM object model. In addition, it describes the RRBC support for DirectToSOM C++ in detail, along with how you can use DirectToSOM C++ to implement classes whose instances objects can be shared across a network using DSOM and with other languages such as Smalltalk. The book also provides a brief discussion of several of the CORBA object services. Throughout the book, programming examples support and emphasize the discussion. Source code for all the examples is available on the diskette that accompanies this book.

DirectToSOM C++ Overview

This chapter covers some basic concepts and provides working examples that will help you get started more effectively with DirectToSOM C++.

Defining DirectToSOM C++ Classes

DirectToSOM C++ classes are distinct from native C++ classes. A class is a DirectToSOM class if it inherits from the predefined DirectToSOM class **SOMObject**, defined in the header file `<som.hh>`. This can be achieved in a variety of ways:

- ♦ **#pragma SOMAsDefault(on|off|pop)** can be used to instruct the compiler to implicitly insert **SOMObject** as the base class for all non-local classes while **on** is in effect. For example, in example below, class **A** is a DirectToSOM class, while class **B** is a native C++ class (**pop** returns to the mode before the most recent **SOMAsDefault** pragma occurrence). The **SOMAsDefault** pragma also implicitly includes the header file `<som.hh>` if it has not already been included.

```

#pragma SOMAsDefault(on)

class A {
    int a;
};

#pragma SOMAsDefault(pop)

class B {
    int b;
};

```

- ♦ A command line switch can implicitly insert **#pragma SOMAsDefault(on)** at beginning of the file. For example, with VisualAge C++ for OS/2 or Windows, this switch is **/Ga**.
- ♦ You can use explicit inheritance from **SOMObject** or another **DirectToSOM** class. For example, all three classes A, B, and C here are **DirectToSOM** classes.

```

#include <som.hh>

class A : public SOMObject {
    int a;
};

#pragma SOMAsDefault(on)
class B {
    int b;
};
#pragma SOMAsDefault(off)

class C: private B {
    int c;
};

```

Defining a **DirectToSOM** class using the **SOMAsDefault** pragma is known as *implicit* or *transparent* mode, whereas defining a **DirectToSOM** class by inheriting from another **DirectToSOM** class (including **SOMObject**) is known as *explicit* mode. Details of how to use the pragma and command line switch will be discussed in Chapter 4, *Using DirectToSOM C++*. Note that **DirectToSOM** classes cannot be declared at block scope; they can only be declared at file scope, or nested within a file scope class.

Using DirectToSOM C++ Classes

Once you have defined a **DirectToSOM** class, what can you do with it? You can create **SOM** objects statically or dynamically, as simple objects, arrays, or as embedded members of other classes, or anywhere else that the declaration of a C++ object is valid. Here are some simple examples (the **SOMDefine** pragma will be discussed shortly).

Using DirectToSOM C++ (samples.cpp):

```

1  #include <som.hh>
2
3  // SOM class
4  class OuterSom : public SOMObject {
5  public:
6      int i;
7  private:
8      class InnerCPP {
9          // pointer to SOM object in nested native class
10         OuterSom *ptr;
11     };
12 };
13
14 #pragma SOMAsDefault(on)
15
16 // SOM class
17 class Outer2Som {
18 public:
19     // embedded SOM member in SOM class
20     OuterSom somobj;
21 private:
22     // embedded SOM class
23     class Inner2Som {
24         int i;
25     } somobj2;
26 };
27
28 #pragma SOMAsDefault(off)
29
30 // native class
31 class Outer3Cpp {
32 public:
33     // embedded SOM member in native class
34     Outer2Som somobj;
35 };
36
37 #pragma SOMDefine(OuterSom)
38 #pragma SOMDefine(Outer2Som)
39 #pragma SOMDefine(Outer2Som::Inner2Som)
40
41 // array of SOM classes
42 OuterSom array_of_som[100];
43
44 int main(void)
45 {
46     // automatic SOM class instance
47     Outer2Som outer2_obj;
48
49     Outer3Cpp *ptr = new Outer3Cpp;
50     ptr->somobj.somobj.i = 10;
51
52     // dynamic SOM class instance

```

Continued

```

53     OuterSom *ptr2 = new OuterSom;
54     ptr2->i = 10;
55
56     // pointer to SOM class member
57     int (OuterSom::*pm);
58     pm = &OuterSom::i;
59     (ptr2->*pm) = 15;
60 }

```

Most of the C++ rules and syntax apply to DirectToSOM classes and objects, with some restrictions. Because the size of a SOM object is not known until run time, compile-time constant expressions such as `sizeof` are treated as run-time constant expressions. Such operators can still be used with SOM objects, but not in contexts that require compile-time evaluation. Such restrictions are discussed in Chapter 5, *Programming Considerations*.

A DirectToSOM C++ class has a SOM release order that by default will contain all member functions and static data members introduced by the class, including those with private and protected access, in the order of declaration. In general, virtual function overrides do not appear in this list, but will appear in the release order for the introducing class. Using the default, you must add any new member functions or static data members at the end of the class. Instead of relying on declaration order, you can instead use a pragma to specify the release order, in which case you can add new release order elements anywhere in the class, but you must add their names to the end of the list.

C++ instance data members in a DirectToSOM class are regrouped into contiguous chunks according to access, in the order of declaration within the class. This regrouping gives efficient access to data members from client code, while enabling RRBC. The location of each chunk is determined at run time through the SOM API. If the declaration order of public and protected data within a class is not changed, and new members are added after any preexisting members of the same access, this scheme allows new data members to be added without requiring recompilation of any code outside the class.

All DirectToSOM C++ class function and data members access is performed through the SOM API, rather than the statically defined compiler constructs used by standard C++. This provides for both RRBC and an implementation-independent object model. The release order and instance data regrouping is covered in more detail in Chapters 3 and 6, *Release-to-Release Binary Compatibility* and *Inside DirectToSOM C++*.

Basic Concepts

This section discusses some basic concepts that are important to understand when working with DirectToSOM C++.

Inheritance

SOM does not support an inheritance tree containing anything other than SOM classes. For DirectToSOM C++, this implies that a class hierarchy must contain all SOM or all native C++ classes; a mixed hierarchy is not supported. SOM also does not permit multiple subobjects of the same type within an inheritance tree. The corresponding DirectToSOM rule is that a class may appear multiple times within a hierarchy only as a virtual base. In other words, only a single occurrence of each nonvirtual base class is allowed within a SOM hierarchy. The compiler will issue a warning for each multiple occurrence of a nonvirtual base class in a SOM class hierarchy. **SOMobject** is a special case, as it is implicitly treated as a virtual base. To illustrate these restrictions, the following shows various combinations of valid and invalid class hierarchies:

```

1 #include <som.hh>
2
3 class Som1 : public SOMObject {
4 };
5
6 class Som2 : public SOMObject {
7 };
8
9 // valid hierarchy: all SOM classes
10 class Som3: private Som1, protected Som2 {
11 };
12
13 // valid hierarchy: all SOM classes
14 class Som4: virtual public Som1,
15           virtual private Som2 {
16 };
17
18 class nonSom1 {
19 };
20
21 // invalid hierarchy: mixing SOM and native
22 class mixed : public nonSom1, private Som1 {
23 };
24
25 // invalid hierarchy: Som2 non-virtual
26 // and appears twice
27 class Som5 : private Som3, public Som2 {
28 };
29
30 // invalid hierarchy:
31 // Som2 non-virtual in Som3, so appears twice
32 class Som6 : private Som3,
33           virtual public Som2 {
34 };
35
36 // valid hierarchy:

```

Continued

```

37 // Som2 virtual in all bases
38 class Som7 : protected Som4,
39             virtual public Som2 {
40 };

```

SOM Class Data Structures

For each SOM class implementation, the DirectToSOM C++ compiler must generate several data structures and export three symbols for use by the SOM run time. The exported symbols are **<class>ClassData**, **<class>C-ClassData**, and **<class>NewClass**. (The meaning and use of these symbols will be explained in detail in Chapter 6 *Inside DirectToSOM C++*.) But the compiler should generate these structures and symbol exports only once per class implementation, otherwise there would be wasted storage and possible duplicate declaration problems. This is certainly a sensible rule; the problem is determining when to generate the structures. In other words, how does the compiler determine, when parsing a SOM class definition, whether the implementation or the client code is being compiled?

For classes that have at least one out-of-line function, the implementation is defined as the file where the definition of the first nonstatic out-of-line member function is defined. The compiler generates the SOM class data structures and symbol exports as part of compiling this file. This ensures that the class data structures are only defined once for that class implementation. In the following example, the SOM class data structures for class A will be generated with the file that contains the definition for the member function show.

```

1 #include <som.hh>
2
3 class A : public SOMObject {
4 private:
5     int i;
6 public:
7     A() { i = 0; }
8     void show();
9     void set_i(int newvalue) { i = newvalue; }
10    int get_i() { return i; }
11 };

```

However, for classes that have all inline or no member functions, there is no way for the compiler to determine where to generate the structures. In such cases, you must explicitly indicate where the structures should be generated using the **SOMDefine** pragma, which is why the **SOMDefine** pragmas are used in the earlier example. Without them, the compiler would not generate the SOM class data structures for classes `OuterSom`, `Outer2Som`, and `Outer2Som::Inner2Som`. This would result in link errors due to unresolved implicit references to these structures in the function `main`. (See

Chapter 4 *Using DirectToSOM C++* for details about the syntax and use of this pragma.)

The compiler will generate the SOM class data structure and symbol exports each time it encounters this pragma for a given class. It is therefore not a good idea to include the pragma with the class definition, but in a separate file. Otherwise, the compiler will generate the structures each time the header file is parsed, resulting in wasted storage and possible duplicate definition link errors.

Although the preceding discussion may seem somewhat irrelevant, it is important to understand how DirectToSOM C++ classes are defined. Duplicate definitions or no definitions for the SOM class data structures are the most common, and typically among the first, programming problems encountered when using DirectToSOM C++.

Linking

As part of creating the SOM class data structures, the address of each function and static data member in the class is supplied to SOM by the DirectToSOM C++ compiler. This implies that all function and static data members must be defined by link time because there are external references to them. If you don't supply definitions for all such members, unresolved reference errors will occur at link time. This is different from native C++, where you don't need to define a member unless it is explicitly referenced in the program. If you simply turn SOM mode on for a given class, and attempt to create a library, you may discover that some methods are missing implementations that would not have mattered in native C++.

Default Constructor

You should always supply a default constructor for a DirectToSOM class. While you may not use this constructor explicitly in your application, many of the SOM frameworks, such as DSOM, require that one be present. In addition, SOM programs written using other languages typically depend upon a default constructor being available. If you are working strictly within DirectToSOM C++, and not using any of the frameworks, then technically you don't need to supply it. However, it's best to get in the habit to avoid bugs later on. While the SOM RRBC support makes it easy to add one if needed, run-time errors caused by a missing default constructor can sometimes be difficult to track down.

Header Files

As you have probably noticed, the DirectToSOM C++ header files used so far all have a file extension of **.hh**. You should always put DirectToSOM C++

classes in a file with an **.hh** extension. This is not simply a convention; it is required for IDL generation.

Also, with regard to header files, you cannot mix the C++ bindings **.xh** header files with the DirectToSOM C++ **.hh** header files in a single compilation unit. The reason is that each provides different definitions of classes such as **SOMObject**. Note that although you can mix programs compiled with these different headers at run time and share SOM objects between them, you cannot mix them at compile time.

Name Mangling

SOM is case-insensitive, so all names presented to it must be unique without respect to case; in particular, class names cannot differ only by case. In order to ensure that unique DirectToSOM C++ names are also unique in SOM, class and member names are subject to a case-insensitive conversion:

- ♦ Uppercase letters are converted to the lowercase equivalent, prepended by lowercase **z**.
- ♦ **z_** is used to mean lowercase **z**.

Thus, **Hello** becomes **zhello** and **ZebraClassZz** becomes **zzebraz-classzzz_**.

This converted name is known as the SOM name, as opposed to the C++ name. For example, **zhello** is the default SOM name for the C++ class named **Hello**. Name mangling will be discussed in more detail in Chapters 6 and 7 *Inside DirectToSOM C++* and *IDL Generation*, but as you start working with DirectToSOM, you will probably notice this mangling taking place.

SOMObject Methods

The **SOMObject** base class from which all DirectToSOM C++ classes derive defines 10 special methods to which certain C++ methods are mapped. This mapping will be discussed in more detail throughout the book, specifically in Chapter 6 *Inside DirectToSOM C++*. But it is worth knowing that this mapping is taking place, particularly because the mapped C++ methods are considered overrides of the **SOMObject** methods, rather than newly introduced methods in the class. For a given class **x**, C++ methods, if supplied, are mapped to the 10 special SOM object methods as follows:

<code>X()</code>	somDefaultInit
<code>~X()</code>	somDestruct
<code>X(X&)</code>	somDefaultCopyInit
<code>X(X const &)</code>	somDefaultConstCopyInit
<code>X(X volatile &)</code>	somDefaultVCopyInit
<code>X(X const volatile &)</code>	somDefaultConstVCopyInit


```

operator=(X&)
operator=(X const &)
operator=(X volatile &)
operator=(X const volatile &)

```

somDefaultAssign
somDefaultConstAssign
somDefaultVAssign
somDefaultConstVAssign

Metaclasses

SOM supports a model similar to that of Smalltalk in that classes are not purely syntactic entities, as in C++, but are themselves objects. SOM class objects are created at run time as required by the client, and are used for creating and manipulating instances. Class objects support a variety of methods for creating and querying objects, such as determining the size of class instances, whether a method is supported by a given class, and whether a given object is a member of that class.

Figure 2.1 illustrates this model. As shown at the top of the figure, a native C++ class is a syntactic entity whose definition is compiled into the program object. A native C++ class has no representation outside of the source code that defines it. With the SOM model, however, as shown at the bottom of the figure, a SOM class is also an object that exists at run time. Each SOM class object is an instance of a special class, called a *metaclass*, which by

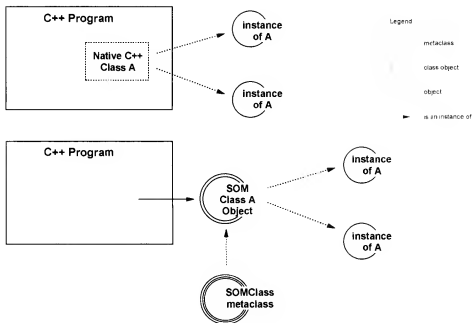


FIGURE 2.1 SOM class object.

default is the class **SOMClass**. In the same way that a class defines the behavior of its instances, a metaclass defines the behavior of its instance, which are class objects. For example, the class **A** may define the method `foo`, in which case `foo` can be invoked against all instance of **A**. Class object methods typically deal with the creation and destruction of class objects. For example, **SOMClass** defines methods such as **somNew**. Since the class object **A** is an instance of the class **SOMClass**, the method **somNew** may be invoked against that class object in order to create a new instance of **A**.

If this is the first time you have been exposed to the concept of a metaclass, it may seem a little strange at first compared to native C++. But it's not that complicated a concept—it's really just a difference in how things are done in the model (although the metaclass concept is much more flexible than the native C++ model). For example, with native C++, you can create instances by invoking the **new** operator, which is applied to a class name. With SOM, the corresponding operation is to invoke the method **somNew** against the appropriate class object. When working with SOM you do not need to deal with metaclasses very often, but it helps to understand the concept.

Setup

Before you can use DirectToSOM C++, you must first install the SOMObjects Developer Toolkit on your system, according to the instructions that come with the Toolkit documentation. DirectToSOM C++ depends upon several header files that are included from the header file `<som.hh>`. This file is shipped in the include directory for the Toolkit, but the other files are not, so you may need to generate them if this is the first time that you are using DirectToSOM C++. In OS/2 or Windows, you can generate the DirectToSOM C++ header files for all the shipped IDL files by running the toolkit command **somhh**, or by issuing the following command in the Toolkit include directory:

```
sc -shh -qmnqualifytypes -musexhpass *.idl
```

This invokes the SOM compiler, which compiles the IDL files and emits the corresponding DirectToSOM C++ files (as requested by the **-shh** parameter). IDL and the SOM compiler will be discussed in more detail in Chapter 7, *IDL Generation*.

The preceding command will create DirectToSOM C++ files for all the frameworks included with SOM, and may take several minutes. Alternatively, you can generate just the files necessary for basic DirectToSOM C++ development, which are `<somobj.hh>`, `<somcls.hh>`, and `<somcm.hh>`. These can be generated from the supplied corresponding IDL files using the

sc command, specifying the individual file names. (You must run this command from the Toolkit include directory.) For example:

```
sc -shh -qmnqualifytypes somobj.idl
```

Once you have the SOM Toolkit installed and the header files set up, you should be able to compile and run the simple DirectToSOM C++ example shown next. Make sure that the Toolkit header files can be found in the include path for the compiler ahead of any other header files, including the compiler library header files. With VisualAge C++ for OS/2 or Windows, you would compile the program with the following command:

```
icc main.cpp hello.cpp -I%SOMBASE%\include
```

Alternatively, you could set the **INCLUDE** environment variable to specify the Toolkit header file directory.

By default, the VisualAge C++ compilers for OS/2 and Windows link any programs containing DirectToSOM C++ classes with the **somtk.lib** library. If you are using another system, you may or may not need to link this library explicitly. See your compiler documentation for details.

Definition of DirectToSOM C++ Class Hello (hello\hello.hh):

```
1 #include <som.hh>
2
3 class Hello : public SOMObject {
4     public:
5         void sayHello();
6 };
```

Implementation of DirectToSOM C++ Class Hello (hello\hello.cpp):

```
1 #include <iostream.h>
2 #include "hello.hh"
3
4 void Hello::sayHello()
5 {
6     cout << "Hello world" << endl;
7 }
```

Client of DirectToSOM C++ Class Hello (hello\main.cpp):

```
1 #include "hello.hh"
2
3
4 int main()
5 {
6     Hello obj;
7
8     obj.sayHello();
9 }
```

A Complete Example

To give you a feel for the flexibility of DirectToSOM C++, the following complete example shows a program that implements and manipulates a message queue. This example will be extended and used in a variety of contexts throughout the book.

The program builds and maintains lists of messages, or message queues. A message queue is described by the SOM class `MessageQueue`, which is defined in the header file `mqueue.hh`. `MessageQueue` is a DirectToSOM C++ class through explicit inheritance from **SOMObject**. (The `_declspec` specifier at line 12 is required for Windows to import data from a DLL. This will be discussed in more detail in Chapter 5, *Programming Considerations*, in the section **SOM DLLs**.) Each message queue has a character string name (data member `name`), and supports methods to send messages to and receive messages from the queue, clear the queue, and dump the contents of the queue. The `MessageQueue` class has two constructors, a default constructor that accepts no arguments, and one that accepts a character string for the queue name. (The default constructor is not used in this program, but is supplied for SOM, as discussed earlier.)

The nested native C++ class `MessageQueue::Mqueue`, line 16, is used to maintain a last in, first out (LIFO) linked list of message queue contents. This class has two data members, one to point to the next element in the list (`next`), and the other to hold the message text (`message`). A `MessageQueue::Mqueue` object can only be constructed by supplying the message text for that element.

The native C++ class `MessageQueueManager` in file `mqmgr.h` is used to manage a list of message queues. It maintains the message queues through an array of pointers to message queues (data member `mqueues`), and supports methods to retrieve a message queue either by name or by number. Retrieval by number allows an application to iterate through the message queue list. Note that `MessageQueueManager` is a native C++ class that declares an array of SOM objects as a data member. Figure 2.2 illustrates how the various classes in the message queue program are related.

The implementation for the classes `MessageQueue::Mqueue` and `MessageQueue` is shown in `mqueue.cpp`. The constructor for `MessageQueue::Mqueue` at line 5 creates a new message queue element by allocating storage for the supplied string parameter and copying the value in. The destructor for that class at line 13 deletes the allocated message storage and recursively deletes all elements in the chain of messages.

The nondefault constructor for `MessageQueue` at line 26 allocates storage for the message queue name and initializes the other data members. The destructor at line 34 deletes the allocated queue name storage and the message queue itself by invoking the `MessageQueue::Clear` method, which is shown at line 81. The `Send` method at line 41 appends a new message queue element to the end of the message queue, while the `Receive` method

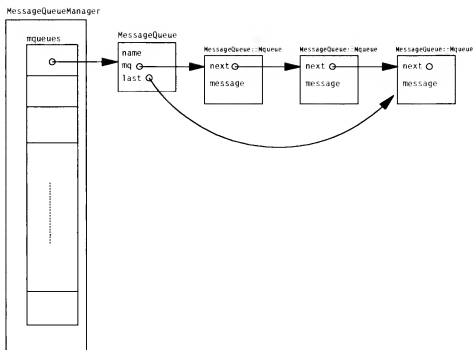


FIGURE 2.2 Message queue data structure diagram.

at line 55 removes and returns the first message in the queue. The `Dump` method at line 71 displays the message queue contents to standard output, while the `clear` method clears the message queue contents.

The implementation for the class `MessageQueueManager` is shown in `mqmgr.cpp`. The constructor, at line 4, initializes each message queue pointer in the array `mqueues` to `NULL`, while the destructor, at line 10, deletes each message queue element. The method `GetMessageQueue(char *)` at line 17 searches for a message queue with the requested name. If a matching queue is found, that queue is returned. Otherwise, if there is space, a new `MessageQueue` is created, stored in the array, and returned. The method `GetMessageQueue(int)` at line 35 returns a message queue by its ordinal position in the array. This method is supplied to allow a client to iterate through the existing message queues.

`tstmq.cpp` is a driver program that uses the message queue. The main program consists mainly of a loop, from lines 10 through 65. When the program is run, the user is prompted to enter a choice and, optionally, a message queue name and a message. The program loops continuously until the user chooses to exit.

Note the call to **SOM_CreateLocalEnvironment** at line 7. `_SOMEnv`, to which the result is assigned, is a compiler-generated variable that is implicitly

passed to all SOM methods, for error handling purposes. Line 7 allocates storage for and initializes this structure, while line 66 deallocates this structure. *_SOMEnv* will be discussed in more detail in Chapter 5, *Programming Considerations*, in the section *Environment Parameter and Error Handling*.

At line 8, a *MessageQueueManager* object, *mqlist*, is declared. This object is used throughout the program, such as at line 23, to retrieve a *MessageQueue* pointer. Each time a request for a particular message queue is made by queue name, the *MessageQueueManager::GetMessageQueue(char *)* method is invoked to retrieve a pointer to the message queue. This will return an existing message queue, a newly created one, or NULL if a new message queue could not be created. At line 43, the *MessageQueueManager::GetMessageQueue(int)* method is used to iterate through the message queues and display their names.

And that's it for the program itself. As you can see, you can write *DirectToSOM* C++ programs with very little specialized SOM code. Most of this example consists of just straight C++ and standard library calls. If you use some of the more advanced features of SOM, such as Distributed SOM, you will need to specify a little more SOM-specific information, but this is typically restricted to the class definition. The implementation and client usually don't require much, if any, special code.

The makefiles for OS/2 and Windows are shown on page 34 following the source code. The *mqueue.cpp* file is compiled into the *mqueue.dll* library, while the *tstmq.cpp* and *mqmgr.cpp* files are compiled together into the *tstmq.exe* program. The SOM compiler *sc* command is used to generate the *mqueue.def* file from the *mqueue.idl* file. This latter file contains the IDL definition for the class, which is generated by the *DirectToSOM* compiler by compiling the *mqueue.hh* file. See Chapter 5, *Programming Considerations*, the section *SOM DLLs* for further details about how SOM DLLs are created, and Chapter 7, *IDL Generation* for more information about generating IDL from a *DirectToSOM* class definition.

Definition of *DirectToSOM* C++ Class *MessageQueue* (file *mqueue\mqueue.hh*):

```

1  #ifndef MQUEUE_H
2  #define MQUEUE_H
3
4  #include <som.h>
5
6  #define SUCCESS 0
7  #define FAIL 1
8  #define MAX_QUEUE_NAME_LEN 20
9  #define MAX_MESSAGE_LEN 256
10
11 #ifdef WINCLIENT
12 _declspec(dllimport)
13 #endif

```

```

14 class MessageQueue : public SOMObject {
15     // keeps a linked list of message entries
16     struct Mqueue {
17         Mqueue *next;
18         char* message;
19         Mqueue(char *);
20         ~Mqueue();
21     };
22     // pointers to first and last elems in queue
23     Mqueue *mq, *last;
24 public:
25     // queue name
26     char *name;
27
28     MessageQueue();
29     // construct with queue name
30     MessageQueue(char *name);
31     ~MessageQueue();
32
33     // clear/delete all messages from the queue
34     virtual void Clear();
35     // send a message to the queue
36     virtual int Send(char *msg);
37     // receive/delete a message in LIFO
38     virtual int Receive(char *msg);
39     // dump the queue contents
40     virtual void Dump();
41 };
42
43 #endif

```

**Definition of Native C++ Class MessageQueueManager
(file *mqqueue\mqmgr.h*):**

```

1  #include "mqqueue.hh"
2
3  #ifndef MQSERVER_H
4  #define MQSERVER_H
5
6  #define MAX_QUEUES 20
7
8  class MessageQueueManager {
9      // list of message queues
10     MessageQueue *mqueues[MAX_QUEUES];
11 public:
12     MessageQueueManager();
13     ~MessageQueueManager();
14     // retrieve queue by name
15     MessageQueue *GetMessageQueue(char *);
16     // retrieve queue by number
17     MessageQueue *GetMessageQueue(int);
18 };
19
20 #endif

```

***Implementation of DirectToSOM C++ Class MessageQueue
(file mqueue\mqueue.cpp):***

```

1  #include <iostream.h>
2  #include <assert.h>
3  #include "mqueue.hh"
4
5  MessageQueue::MQueue::MQueue(char *elemMessage)
6  {
7      next = NULL;
8      message = new char[strlen(elemMessage) + 1];
9      assert(message != NULL);
10     strcpy(message, elemMessage);
11 }
12
13 MessageQueue::MQueue::~MQueue()
14 {
15     delete message;
16     if (next)
17         delete next;
18 }
19
20 MessageQueue::MessageQueue()
21 {
22     name = NULL;
23     last = mq = NULL;
24 }
25
26 MessageQueue::MessageQueue(char *qname)
27 {
28     last = mq = NULL;
29     name = new char[strlen(qname) + 1];
30     assert(name != NULL);
31     strcpy(name, qname);
32 }
33
34 MessageQueue::~MessageQueue()
35 {
36     Clear();
37     if (name)
38         delete name;
39 }
40
41 int MessageQueue::Send(char *message)
42 {
43     MQueue *elem;
44     if (! (elem = new MQueue(message)))
45         return FAIL;
46     if (mq == NULL) {
47         mq = last = elem;
48     } else {
49         last->next = elem;
50         last = elem;
51     }

```



```

52     return SUCCESS;
53 }
54
55 int MessageQueue::Receive(char *buf)
56 {
57     if (!mq) {
58         return FAIL;
59     }
60     Mqueue *elem = mq;
61     mq = mq->next;
62     if (last == elem)
63         last = NULL;
64     strcpy(buf, elem->message);
65     // so don't delete entire chain
66     elem->next = NULL;
67     delete elem;
68     return SUCCESS;
69 }
70
71 void MessageQueue::Dump()
72 {
73     int i = 1;
74     cout << "Dumping queue " << name << endl;
75     for (Mqueue *cur = mq; cur != NULL;
76          cur=cur->next, i++)
77         cout << '\t' << i << ": "
78              << cur->message << endl;
79 }
80
81 void MessageQueue::Clear()
82 {
83     if (mq != NULL) {
84         delete mq;
85         mq = last = NULL;
86     }
87 }

```

**Implementation of Native C++ Class MessageQueueManager
(file mqueue\mqmgr.cpp):**

```

1  #include <iostream.h>
2  #include "mqmgr.h"
3
4  MessageQueueManager::MessageQueueManager()
5  {
6      for (int i=0; i<MAX_QUEUES; i++)
7          mqueues[i] = NULL;
8  }
9
10 MessageQueueManager::~MessageQueueManager()
11 {
12     for (int i=0; i<MAX_QUEUES; i++)
13         if (mqueues[i])
14             delete mqueues[i];

```

Continued

```

15 )
16
17 MessageQueue *MessageQueueManager::
18 GetMessageQueue(char *name)
19 {
20     for (int i=0; i<MAX_QUEUES; i++)
21         if (mqueues[i] &&
22             mqueues[i]->name &&
23             strcmp(mqueues[i]->name, name) == 0)
24             return mqueues[i];
25     for (i=0; i<MAX_QUEUES && mqueues[i]; i++)
26         ;
27     if (i == MAX_QUEUES)
28         return NULL;
29     if (! (mqueues[i] = new MessageQueue(name))) {
30         return NULL;
31     }
32     return mqueues[i];
33 }
34
35 MessageQueue *MessageQueueManager::
36 GetMessageQueue(int qnum)
37 {
38     if (qnum < MAX_QUEUES && mqueues[qnum])
39         return(mqueues[qnum]);
40     return NULL;
41 }

```

MessageQueue Driver Program (file mqueue\stmq.cpp):

```

1 #include <iostream.h>
2 #include "mqmgr.h"
3
4 int main(int argc, char *argv[])
5 {
6     // _SOMEnv is compiler-generated
7     _SOMEnv = SOM_CreateLocalEnvironment();
8     MessageQueueManager mqmgr;
9
10    for ( ;; ) {
11        char choice, qname[MAX_QUEUE_NAME_LEN],
12            message[MAX_MESSAGE_LEN];
13        MessageQueue *mq;
14        cout << "Enter choice (S)end, (R)ecieve, "
15              "(L)ist, (D)ump, (C)lear, (Q)uit: ";
16        cin >> choice;
17        switch (choice) {
18
19            case 's': case 'S':
20                cout << "Enter queue name and message: ";
21                cin >> qname;
22                cin.getline(message, sizeof(message));
23                if ( (mq=mqmgr.GetMessageQueue(qname)) != NULL)

```

```

24         mq->Send(message);
25         break;
26
27     case 'r': case 'R':
28         cout << "Enter queue name: ";
29         cin >> qname;
30         if ( (mq=qlist.GetMessageQueue(qname)) != NULL) {
31             if (mq->Receive(message) == SUCCESS)
32                 cout << "Received message from queue " <<
33                     qname << ": " << message << endl;
34             else
35                 cout << "No message from from queue "
36                     << qname << endl;
37         }
38         break;
39
40     case 'l': case 'L':
41         int i;
42         for (i=0; i < MAX_QUEUES; i++) {
43             if ( (mq=qlist.GetMessageQueue(i)) != NULL)
44                 cout << "Name: " << mq->name << endl;
45         }
46         break;
47
48     case 'd': case 'D':
49         cout << "Enter queue name: ";
50         cin >> qname;
51         if ( (mq=qlist.GetMessageQueue(qname)) != NULL)
52             mq->Dump();
53         break;
54
55     case 'c': case 'C':
56         cout << "Enter queue name: ";
57         cin >> qname;
58         if ( (mq=qlist.GetMessageQueue(qname)) != NULL)
59             mq->Clear();
60         break;
61
62     case 'q': case 'Q':
63         return 0;
64 }
65 }
66 SOM_DestroyLocalEnvironment(_SOMEnv);
67
68 return(0);
69 }
70

```

OS/2 Makefile (file mqueue\makefile.os2):

```

1 #
2 # Makefile for OS/2
3 #
4

```

Continued

```

5 all: mqueue.def mqueue.dll tstmq.exe
6
7 # imblib creates mqueue.lib
8 mqueue.dll: mqueue.hh mqueue.cpp
9     gcc /Ti+ /Ge- /B*/NOE* mqueue.cpp mqueue.def
10     implib mqueue.lib mqueue.dll
11
12 tstmq.exe: tstmq.cpp mqmgr.cpp
13     gcc /Ti+ tstmq.cpp mqmgr.cpp mqueue.lib
14
15 # gcc creates mqueue.idl
16 # sc creates mqueue.def
17 mqueue.def: mqueue.hh
18     gcc mqueue.hh
19     sc -sdef mqueue.idl

```

Windows Makefile (file *mqueue\makefile.win*):

```

1 #
2 # Makefile for Windows
3 #
4
5 all: mqueue.def mqueue.dll tstmq.exe
6
7 # ilib creates mqueue.lib file
8 mqueue.dll: mqueue.hh mqueue.cpp
9     gcc /Ti+ /Ge- /B*/NOE* mqueue.cpp mqueue.def
10     ilib /geni mqueue.def
11
12 tstmq.exe: tstmq.cpp mqueue.hh mqmgr.cpp
13     gcc /Ti+ -DWINCLIENT \
14         tstmq.cpp mqmgr.cpp mqueue.lib
15
16 # gcc creates mqueue.idl file
17 # sc creates mqueue.def file
18 mqueue.def: mqueue.hh
19     gcc mqueue.hh
20     $(SOMBASE)\bin\sc -sdef mqueue.idl

```

Release-to-Release Binary Compatibility

This chapter describes the release-to-release binary compatibility (RRBC) capability provided by DirectToSOM C++, detailing the supported and unsupported changes for updating class implementations in a binary compatible way.

Supporting RRBC

One of the major underpinnings of the SOM support for RRBC is the concept of a *release order*. Every class has a release order, which defines the order in which member functions and static data members introduced by the class are released from that class. SOM maintains binary compatibility by assigning each member function and static data member a specific location in the release order list; clients locate these members by their position within the releaser order list. As long as the order remains invariant, RRBC is maintained.

The release order maps to a table exported by the class implementation called the **<className>ClassData** structure; member functions and

static data members introduced by the class are accessed by clients through their release order slot in the exported table. In order to maintain RRBC for a class, the release order must remain invariant, with the exception that new members can be added to the end of the release order list. By default, the release order for a class is assumed to be the order in which these members appear within the class definition. New members must be added after all existing ones in order to maintain RRBC. You can also explicitly supply the release order through the **SOMReleaseOrder** pragma, and can then add members anywhere in the class and simply add the name to the end of release order list.

For example, the corresponding release orders for the two classes shown in the programming example at the bottom of this page are:

release order for A:

```
1:a2
2:foo()
3:bar()
4:bar(int)
```

release order for B:

```
1:f2()
2:f1()
3:f2(int)
4:b2
```

A client accessing `foo()` introduced in class A will use slot 2 in the `AClassData` structure, while a client accessing `f2(int)` introduced in class B will use slot 3 in the `BClassData` structure (this is all handled implicitly by the `DirectToSOM` compiler). For class A, any new release order members must be defined in the class after all others; that is, after `bar(int)`. For class B, new members can be defined anywhere in class, and the name simply added to the end of the **SOMReleaseOrder** pragma list. Note that you cannot reorder the release order or delete a member from it, because clients are dependent upon the released ordering.

Release Order (ro1.hh):

```
1 #include <som.hh>
2
3 #pragma SOMAsDefault(on)
4
5 class A {
6     int a1;
7     static int a2;
8     void foo();
9     public:
10     int a3;
11     int a4;
12     virtual void bar();
13     virtual void bar(int);
14     private:
15     int a5;
16 };
17
```

```

18 class B {
19     int b1;
20     static int b2;
21     void f1();
22 public:
23     int b3;
24     int b4;
25     virtual void f2();
26     virtual void f2(int);
27 private:
28     int b5;
29     #pragma SOMReleaseOrder(f2(), f1, f2(int), b2)
30 };

```

The **<className>ClassData** slots don't contain direct function pointers to the target members: rather, they are pointers to *thunks*, which are small pieces of code created and updated by the SOM run-time. The thunk contains code to invoke the target method. When the SOM run-time constructs a class object, it defines a corresponding method table for that class, which it uses to look up the address of a target method from the thunk.

The class object and method table are constructed using information provided by the class implementation, such as the names of parent classes and introduced methods. When a method is invoked against a particular object, the thunk in the **ClassData** table for the introducing class is called, which in turn determines the appropriate method to call using the class method table, which can be accessed from the class instance. More on thunks in Chapter 6, *Inside DirectToSOM C++*.

The release order for a class will contain all static data members and function members that were *introduced* by the class. For most member types, a declaration in a class constitutes an introduction in that class. However, for virtual function members, the introducing class is the first base class to declare that member. As an example, consider the class *c* shown in the programming example on page 38, which inherits from class *a* preceding.

The release order for class *c* is:

```

1:     foo()
2:     foo2()

```

This is because the methods *bar()* and *bar(int)* are virtual, so they are considered introduced by class *a*, whereas *foo()* is a nonvirtual override in class *c*, and *foo2()* is newly introduced. This model provides support for polymorphism: when the method *bar()* is invoked against an instance of class *c*, slot 3 in the *AClassData* structure is used, and the thunk uses the method table for class *c* to look up and invoke the appropriate method *C::bar()*.

Overriding Base Class Methods (ro2.hh):

```

1  #include "ro1.hh"
2
3  class C : public A {
4      int c1;
5      void foo();
6      void foo2();
7  public:
8      int c2;
9      void bar();
10     void bar(int);
11 };

```

In addition to the release order, nonstatic data members declared in each class are reordered into chunks by access—first public, then protected, and then private. When accessing such data members, the DirectToSOM compiler retrieves the chunk address for the particular access type at run time and locates the member by using its offset within the chunk. This allows each chunk to grow separately without affecting class clients or subclasses. (The private and protected data are actually allocated as a single contiguous chunk, protected first, then private. Implementation code obtains addressability to the private data by an offset from the address of the protected data chunk.)

For example, the data layout for an instance of classes A and C is as follows:

<i>instance data order for A:</i>		<i>instance data order for C:</i>	
public from A:	a3	public from A:	a3
	a4		a4
private from A:	a1	private from A:	a1
	a5		a5
		public from C:	c2
		private from C:	c1

As long as new data members are added at the end of their access group (after a4 in A and c2 in C for public, and after a5 in A and c1 in C for private), data members can be added to the class without requiring recompilation of client code. As with the release order, removing a data member would break RRBC because clients are dependent upon certain data members being at a constant offset.

Instances of DirectToSOM classes are implemented by the compiler as references to hidden instance structures. This approach allows a DirectToSOM instance to act like a standard class instance in most situations. For example, you can declare DirectToSOM instances with automatic or static storage duration, or allocate them dynamically, and declare them as

structure members of both DirectToSOM and native C++ classes. The restrictions of this implementation are discussed in Chapter 5, *Programming Considerations*.

Supported Changes

This section covers the types of changes that you can make to a DirectToSOM C++ class without breaking RRBC. The basic RRBC rule with DirectToSOM C++ is that if you make a change to an implementation that does not require a corresponding change in the client, then you don't need to recompile that client.

You can add members to a class according to the following guidelines:

- ♦ *Nonstatic data members:* You must add them after all existing data members of the same access level. For example, given classes A and C shown earlier, new data members must be added after `a4` in A and `c2` in C for public members and after `a5` in A and `c1` in C for private members. Note that private data members can be added anywhere if you will be recompiling the entire class implementation and any friends. The preceding guideline allows you to recompile only the SOM class implementation file (that is, the file where the SOM class data structures are generated) and any methods that depend upon the new data member.
- ♦ *Static data members:* If you are using **SOMReleaseOrder** pragma, you can declare new members anywhere in the class and then add the name to the end of the release order list; otherwise, you must declare them after other static data and member functions. For example, with the classes A and B shown earlier, a new static data member could be added anywhere in class B and added to the end of the release order list, but it would have to be added after `bar(int)` in class A.
- ♦ *Nonvirtual function members:* The same rules apply as for static data members. Note that if you add a member function that hides a nonvirtual base class member, existing clients will continue to call the base class version until they are recompiled.
- ♦ *Virtual function members:* The same rules apply as for static data members. If you override a virtual function defined in a base class, unqualified references to that function through a derived class object will call the new version, but qualified references, such as `derived::f()`, will continue to refer to the base class version until the client is recompiled. Qualified references to the base class function, such as `base::f()`, will always call the base class version.

You can make the following changes to a class:

- ♦ *Promote member functions:* You can promote a member function up the class hierarchy so that it is introduced in a base class. To achieve

this, you must use the **SOMReleaseOrder** pragma on the derived class side and preface that member function with an exclamation point to indicate that the slot should be reserved, but the method is now introduced in the parent class. Existing clients will continue to call using the derived class slot to get the method, which is why it must be reserved, while newly compiled clients will use the parent table slot; but in both cases, the base class method would be called. For example, you could promote the member `foo2()` in class `C` to class `A` as shown next. The resulting release orders would be:

<i>Release order for A:</i>	<i>Release order for C:</i>
1: a2	1: foo()
2: foo()	2: A::foo2() migrated
3: bar()	
4: bar(int)	
5: foo2()	

Method Promotion (migrate.hh):

```

1  #pragma SOMasDefault(on)
2
3  class A {
4      int a1;
5      static int a2;
6      void foo();
7  public:
8      int a3;
9      int a4;
10     virtual void bar();
11     virtual void bar(int);
12     void foo2();
13 private:
14     int a5;
15 };
16
17 class C : public A {
18     int c1;
19     void foo();
20 public:
21     int c2;
22     void bar();
23     void bar(int);
24     #pragma SOMReleaseOrder(foo, !foo2)
25 };

```

- ♦ *Change a nonvirtual member function to virtual:* When a member function is changed from nonvirtual to virtual, any qualified or unqualified calls to that method will continue to call the same method in the same class as before. For any derived class that overrides that function member, calls with a derived class object will continue to call the base

class version as if it were still a nonvirtual function member, until that class is recompiled. As with member function promotion, the **SOMReleaseOrder** pragma must be used to reserve the slot in any derived class that overrides that member.

For example, the method `A::foo()` in the previous example could be changed to virtual as shown in the next programming example. The release order for `A` remains the same, but when `C` is recompiled, the release order for `C` becomes:

```
1:    foo() migrated
2:    A::foo2() migrated
```

Note, however, that until the implementation for class `C` is recompiled, the member `C::foo()` will not be treated as a virtual function. In other words, the method call `((A *)&c)->foo()` for object `c` of type `C` will continue to invoke `A::foo()` until the implementation of class `C` is recompiled, at which point such calls will invoke the `C::foo()` version. As with the migration of `foo2()`, client code will continue to use the `CClassData` structure to invoke the method until the client is recompiled, in which case the slot in `AClassData` will be used.

Making a Member Function Virtual (makevirt.hh):

```
1  #pragma SOMAsDefault(on)
2
3  class A {
4      int a1;
5      static int a2;
6      virtual void foo();
7  public:
8      int a3;
9      int a4;
10     virtual void bar();
11     virtual void bar(int);
12     void foo2();
13 private:
14     int a5;
15 };
16
17 class C : public A {
18     int c1;
19     void foo();
20 public:
21     int c2;
22     void bar();
23     void bar(int);
24     #pragma SOMReleaseOrder(!foo, !foo2)
25 };
```

- ♦ *Reorder member functions:* Member functions can be reordered within the class provided that the release order for the class remains invariant. This can only be accomplished using the **SOMReleaseOrder** pragma to keep the release order constant.
- ♦ *Change member function access:* The access of a member function can be changed, for example from private to public, without affecting the release order. Note that reducing the access of a member function could cause errors when the class is recompiled, as clients may no longer be able to access the function.
- ♦ *Delete private member functions:* Private member functions can be deleted, but their original slot must be reserved in order to keep the release order list invariant. This can be achieved by using an asterisk (*) in place of the name in the release order. Note that any class implementation code that accessed these methods would need to be recompiled. For example, the method `foo()` in the first example of class `A` on page 36 could be deleted as the next programming example shows. The resulting release order for class `A` would be:

```

1:    a2
2:    *
3:    bar()
4:    bar(int)

```

This approach can also be used to remove public or protected member functions from the interface, but any clients using those member functions would encounter run-time errors (which may be the desired effect in some situations).

Deleting a Member Function (delfunc.hh):

```

1  #pragma SOMAsDefault(on)
2
3  class A {
4  int a1;
5  static int a2;
6  public:
7  int a3;
8  int a4;
9  virtual void bar();
10 virtual void bar(int);
11 private:
12 int a5;
13 #pragma SOMReleaseOrder(a2, *, bar(), bar(int))
14 };

```

- ♦ *Delete/promote private data members:* Private data members can be deleted or promoted to a parent class, but this requires recompilation of all class methods and friends. The member must be declared in the parent class according to the rules of nonstatic data member addition.

You can make the following changes to the class hierarchy:

- ♦ *Add base classes:* Additional base classes can be specified for a class without requiring a recompilation of any class clients or subclasses. Existing clients will not be aware of the new base class until they are recompiled.
- ♦ *Delete private base classes:* A private base class can be removed without affecting RRBC, but this may require a recompilation of any class implementation code that references those classes explicitly. Public and protected base classes may also be removed, but this would result in run-time errors in any clients that referenced those classes (removing a protected base class would only impact subclasses, while removing a public base class would impact all clients).

In addition to the supported changes I have described, you can also associate a version with a class object, consisting of a major and minor version, using the **SOMClassVersion** pragma. If the major version with which the client was compiled differs from the major version of the implementation, this will result in an error at run time. This support is limited, however, in that the check will only occur when the class object is created, which is once per application. Several compilation units may have incompatible major versions, but only the major version used by the compilation unit that causes the class object to be created will be checked.

Unsupported Changes

If you make the following changes to a DirectToSOM C++ class, you may compromise RRBC:

Data members:

- ♦ *Delete a public or protected data member:* If a protected data member is deleted, this would require recompilation of all class methods, friends, and subclasses only, assuming there were no references to the data member.
- ♦ *Promote a public or protected data member to a base class:* This is equivalent to deleting the member and adding it to the base class.
- ♦ *Rename or change the access of a data member:* This is equivalent to deleting the member and adding it with the new access or name.
- ♦ *Reorder data members:* This is equivalent to deleting the members in the old positions and adding them in the new positions.
- ♦ *Change the size and/or type of data members:* This is a signature issue as well as an RRBC issue. A change such as redefining an integer as a float would certainly break RRBC, but would also likely cause problems when the client were recompiled.

Member functions:

- ♦ *Remove a public or protected function or static data member from the release order:* The release order must remain consistent with earlier implementations. If an asterisk is used in the release order list, a run-time error results from any references to that member.
- ♦ *Rename a public or protected member function:* This is equivalent to deleting the member and adding it with the new name.
- ♦ *Default parameters are bound to the implementation:* If you change a default value in the class header file, existing clients will use the old value, and newly compiled clients will use the new value. There is one exception: default constructors. Consider a class that contains a constructor whose parameters all have default values, and a default constructor is not provided. The compiler builds a default constructor that will call the supplied constructor, and will pass the default value as part of the implementation. For example, when the default constructor for A is called for the following class, A(int) will always be called with whatever default value the implementation was compiled, in this case, 100.

```

1 class A : public SOMObject {
2     int j;
3     public:
4     A(int i=100): j(i) { }
5 };

```

A Complete Example

In order to provide a more realistic example of the RRBC support provided by DirectToSOM C++, I have updated the `MessageQueue` class to add a new data member and corresponding function member as shown next. At line 25, I have added the data member `count`. Following the RRBC rules for adding data members, `count` appears after all data members of the same access level, which is `private` in this case (although it is not necessary in this example, as the entire class implementation will be recompiled).

Next, at line 36, I have introduced a new member function, `Count()`. Because `Count()` is introduced before existing member functions, a **SOM-ReleaseOrder** pragma is necessary to maintain RRBC, which is included in lines 47 through 53. When you are adding a **SOMReleaseOrder** pragma to an existing class, you can request that the compiler generate the pragma for you, rather than typing it yourself. In OS/2 or Windows, you do this through the **/Fr** compiler option, providing the name of a SOM class, for which the release order will be generated to standard output. For example, I used the command:

```
icc -FrMessageQueue mqueue.hh > ro.out
```

to create the compiler-generated release order. Then I copied the generated file into the `mqueue.hh` file and added the `Count()` method to the end of the list.

Note that I could have simply added `Count()` after all the existing methods and avoided using the **SOMReleaseOrder** pragma altogether. I added `Count()` in the middle partly because I prefer not to be restricted by the release order in determining where within the class header the method can be introduced and to provide an example of how to use the compiler-generated release order list.

In the `MessageQueue` implementation, I have added support for tracking the queue count at lines 24, 31, 54, 71, and 96. The new `MessageQueue::Count` method appears at line 75. The `mqueue.dll` can be re-created from the new source, and will continue to work with the existing client, even though I have added a new member function in the middle of the existing member functions and increased the `MessageQueue` instance size. This applies even to the array of `MessageQueue` objects declared in the `MessageQueueManager` class. Each of those array elements is now bigger by an `int` and it still works! Again, the SOM RRBC rule is that if a change in the class implementation does not require a corresponding source code change in the class client, that client does not need to be recompiled. In this example, the new `MessageQueue` instance size will be determined at run time and used to allocate the array storage for the `MessageQueueManager` class. In addition, the new release order is consistent with the client's view, so method resolution at the client side will invoke the appropriate methods using the `MessageQueueClassData` structure.

Eventually, we may want to update the client to take advantage of the new functionality, as shown at the end of the example. At this point—but only at this point—we will need to recompile the client code. Note, however, that the `MessageQueueManager` code has not changed and therefore still would not require recompilation.

Compiler-Generated Release Order (`mqueue\ro.out`):

```
/* MessageQueue */
#pragma SOMReleaseOrder(\
/* 1 */ MessageQueue(char*),\
/* 2 */ Clear(),\
/* 3 */ Send(char*),\
/* 4 */ Receive(char*),\
/* 5 */ Dump())
```

Updated `MessageQueue` Class to Track Message Count (`mqueue\mqueue.hh`):

```
1 #ifndef MQUEUE_H
2 #define MQUEUE_H
```

Continued

```

3
4 #include <som.hh>
5
6 #define SUCCESS 0
7 #define FAIL 1
8 #define MAX_QUEUE_NAME_LEN 20
9 #define MAX_MESSAGE_LEN 256
10
11 #ifdef WINCLIENT
12 _declspec(dllimport)
13 #endif
14 class MessageQueue : public SOMObject {
15     // keeps a linked list of message entries
16     struct Mqueue {
17         Mqueue *next;
18         char* message;
19         Mqueue(char *);
20         ~Mqueue();
21     };
22     // pointers to first and last elems in queue
23     Mqueue *mq, *last;
24     // number of elements in queue
25     int count;
26 public:
27     // queue name
28     char *name;
29
30     MessageQueue();
31     // construct with queue name
32     MessageQueue(char *name);
33     ~MessageQueue();
34
35     // return count of queue elements
36     virtual int Count();
37     // clear/delete all messages from the queue
38     virtual void Clear();
39     // send a message to the queue
40     virtual int Send(char *msg);
41     // receive/delete a message in LIFO
42     virtual int Receive(char *msg);
43     // dump the queue contents
44     virtual void Dump();
45
46     /* MessageQueue */
47     #pragma SOMReleaseOrder(\
48     /* 1 */ MessageQueue(char*),\
49     /* 2 */ Clear(),\
50     /* 3 */ Send(char*),\
51     /* 4 */ Receive(char*),\
52     /* 5 */ Dump(),\
53     /* 6 */ Count())
54 };
55
56 #endif

```


**Updated MessageQueue Implementation to Track Message Count
(mqueue\mqueue.cpp):**

```

1  #include <iostream.h>
2  #include <assert.h>
3  #include "mqueue.hh"
4
5  MessageQueue::Mqueue::Mqueue(char *elemMessage)
6  {
7      next = NULL;
8      message = new char[strlen(elemMessage) + 1];
9      assert(message != NULL);
10     strcpy(message, elemMessage);
11 }
12
13 MessageQueue::Mqueue::~Mqueue()
14 {
15     delete message;
16     if (next)
17         delete next;
18 }
19
20 MessageQueue::MessageQueue()
21 {
22     name = NULL;
23     last = mq = NULL;
24     count = 0;
25 }
26
27 MessageQueue::MessageQueue(char *qname)
28 {
29     last = mq = NULL;
30     name = new char[strlen(qname) + 1];
31     count = 0;
32     assert(name != NULL);
33     strcpy(name, qname);
34 }
35
36 MessageQueue::~MessageQueue()
37 {
38     Clear();
39     if (name)
40         delete name;
41 }
42
43 int MessageQueue::Send(char *message)
44 {
45     Mqueue *elem;
46     if (! (elem = new Mqueue(message)))
47         return FAIL;
48     if (mq == NULL) {
49         mq = last = elem;
50     } else {

```

Continued

```

51         last->next = elem;
52         last = elem;
53     }
54     ++count;
55     return SUCCESS;
56 }
57
58 int MessageQueue::Receive(char *buf)
59 {
60     if (!mq) {
61         return FAIL;
62     }
63     Mqueue *elem = mq;
64     mq = mq->next;
65     if (last == elem)
66         last = NULL;
67     strcpy(buf, elem->message);
68     // so don't delete entire chain
69     elem->next = NULL;
70     delete elem;
71     --count;
72     return SUCCESS;
73 }
74
75 int MessageQueue::Count()
76 {
77     return count;
78 }
79
80 void MessageQueue::Dump()
81 {
82     int i = 1;
83     cout << "Dumping queue " << name << endl;
84     for (Mqueue *cur = mq; cur != NULL;
85          cur=cur->next, i++)
86         cout << '\t' << i << ": "
87              << cur->message << endl;
88 }
89
90 void MessageQueue::Clear()
91 {
92     if (mq != NULL) {
93         delete mq;
94         mq = last = NULL;
95     }
96     count = 0;
97 }

```

Updated Class Client (mqueue\testmq.cpp):

```

1  #include <iostream.h>
2  #include "mqmgr.h"
3

```

```

4  int main(int argc, char *argv[])
5  {
6      // _SOMEnv is compiler-generated
7      _SOMEnv = SOM_CreateLocalEnvironment();
8      MessageQueueManager mqlist;
9
10     for ( ;; ) {
11         char choice, qname[MAX_QUEUE_NAME_LEN],
12             message[MAX_MESSAGE_LEN];
13         MessageQueue *mq;
14         cout << "Enter choice (S)end, (R)eceive, "
15              << "(L)ist, (D)ump, (C)lear, (Q)uit: ";
16         cin >> choice;
17         switch (choice) {
18
19             case 's': case 'S':
20                 cout << "Enter queue name and message: ";
21                 cin >> qname;
22                 cin.getline(message, sizeof(message));
23                 if ( (mq=mqlist.GetMessageQueue(qname)) != NULL)
24                     mq->Send(message);
25                 break;
26
27             case 'r': case 'R':
28                 cout << "Enter queue name: ";
29                 cin >> qname;
30                 if ( (mq=mqlist.GetMessageQueue(qname)) != NULL) {
31                     if (mq->Receive(message) == SUCCESS)
32                         cout << "Received message from queue " <<
33                          qname << ": " << message << endl;
34                     else
35                         cout << "No message from from queue "
36                          << qname << endl;
37                 }
38                 break;
39
40             case 'l': case 'L':
41                 int i;
42                 for (i=0; i < MAX_QUEUES; i++) {
43                     if ( (mq=mqlist.GetMessageQueue(i)) != NULL)
44                         cout << "Name: " << mq->name <<
45                          " Count: " << mq->Count() << endl;
46                 }
47                 break;
48
49             case 'd': case 'D':
50                 cout << "Enter queue name: ";
51                 cin >> qname;
52                 if ( (mq=mqlist.GetMessageQueue(qname)) != NULL)
53                     mq->Dump();
54                 break;
55
56             case 'c': case 'C':
57                 cout << "Enter queue name: ";

```

Continued

```

58         cin >> qname;
59         if ( (mq=qlist.GetMessageQueue(qname)) != NULL)
60             mq->Clear();
61         break;
62
63         case 'q': case 'Q':
64             return 0;
65     }
66 }
67 SOM_DestroyLocalEnvironment(_SOMEnv);
68
69 return(0);
70 )

```

RRBC Usage Considerations

One of the major considerations in deciding when to make a class DirectToSOM is performance. While DirectToSOM C++ gives you the power and flexibility of the SOM RRBC support through the C++ programming language, its use incurs an overhead. Every DirectToSOM method call requires an indirection through the thunk in the **ClassData** structure, and every instance data access results in a function call to retrieve addressability to the data.

Due to the implicit SOM run-time model, DirectToSOM C++ classes are not suitable for defining concrete classes. By definition, a concrete type will “provide run-time and space efficiency comparable to hand-crafted code” and “have minimal dependency on other classes” (Stroustrup, 1991). Also, for efficiency reasons, there is typically a strong dependency between client and implementation code, requiring recompilation of the client if the implementation changes. This is the other end of the spectrum from the SOM RRBC support, where a performance penalty is acceptable in exchange for alleviating the dependency.

What you will probably find, however, is that you will have a certain set of classes that are exposed in your interface, for which you need to maintain RRBC. Likely, you will have many more internal classes that don't require this additional functionality. You can thus mix and match DirectToSOM and non-DirectToSOM C++ classes as the needs of your application dictate.

As with using C++ instead of C, or a high-level language instead of assembler, using DirectToSOM C++ for RRBC support is a programmer productivity versus program efficiency trade-off. If you choose to manage RRBC yourself, you can probably produce faster code, but generally at the cost of language restrictions and increased program management on your part, which can be tedious and error-prone. There are additional advantages to using DirectToSOM over just RRBC support, such as interlanguage object-sharing capability and distributed object support, which I will discuss in subsequent chapters.

Using DirectToSOM C++

This chapter covers the various pragmas and compiler switches that are available for programming with DirectToSOM C++. You can treat this and the following chapter mostly as reference material. I would recommend that you skim through these two chapters initially so that you are familiar with the features and function available, and return later when you need more details as you develop your program.

DirectToSOM C++ Pragmas

This section will explain the syntax and usage of the pragmas available for DirectToSOM C++. Most of the time, when programming with DirectToSOM C++, you will just use standard C++ syntax and the compiler will map the C++ object model to the SOM model implicitly. There are certain situations, however, where additional information must be supplied through the use of pragmas for SOM concepts and constructs that have no counterpart in C++. Basically, the more you want to do, the more you need to do. In other words, the more you want to take advantage of SOM support, such as DSOM, the more information you will need to specify through pragmas.

These pragmas apply almost universally to the class definition, rather than the usage of the class, so you will find that if you do need to

specify any pragmas, their use is typically limited to the class header file. The class implementation and client code is still mostly normal C++. I will provide a brief description of each pragma, followed by a more detailed discussion. Currently, the following pragmas are supported for DirectToSOM C++:

SOM	Turns on DirectToSOM mode for the compilation.
SOMAsDefault	Toggles interpretation of all native classes as SOM or not.
SOMAttribute	Defines a CORBA attribute.
SOMCallStyle	Specifies the callstyle (IDL/OIDL) for methods of the class.
SOMClassInit	Specifies a function to be called when the class object is created.
SOMClassName	Defines the SOM name for a class.
SOMClassVersion	Specifies the version of the class implementation.
SOMDataName	Defines the SOM name for a data member.
SOMDefine	Forces generation of the SOM class data structures for a class.
SOMIDLDecl	Specifies the IDL declaration that should be generated.
SOMIDLPass	Allows arbitrary IDL to be generated with a class.
SOMIDLTypes	Forces the generation of given type with the IDL for a class.
SOMMetaClass	Specifies the metaclass for a class.
SOMMethodAppend	Generates additional information for SOM method.
SOMMethodName	Defines the SOM name for a member function.
SOMModule	Designates a class as an IDL module.
SOMNoDataDirect	Prevents direct access to data members.
SOMNoMangling	Prevents mangling of member names.
SOMNonDTS	Indicates the class is not implemented using DirectToSOM C++.
SOMReleaseOrder	Specifies the release order for members of the class.

Conventions Used with the DirectToSOM Pragmas

Some of the DirectToSOM pragmas can be applied to a specific class only, while others can be applied to a range of classes. In order to indicate the range across which these latter pragmas apply, and to prevent interference between pragmas specified in different header files, the pragma settings are implemented as a stack. You can push a new setting onto the stack, and pop a previous setting off.

This stack mechanism is supported through the keywords of **on**, **off**, or **pop**. For example, the syntax for the **SOMAsDefault** pragma is:

```
#pragma SOMAsDefault(on | off | pop)
```

Specifying **on** will turn **SOMAsDefault** processing on, specifying **off** turns this processing off, and **pop** restores the setting to the previous one. This implementation is particularly useful within a header file. You can, for example, specify **SOMAsDefault(on)** at the beginning of the file, and then specify **SOMAsDefault(pop)** at the end. This way, once this header file has been processed, whatever mode was in place before the file was included will be restored. Without **pop**, you would need a mechanism to query the previous state, then save it, and restore it at end.

Another convention used with the DirectToSOM pragmas is an asterisk to represent the current class in scope. For example, the syntax for the **SOMClassName** pragma is:

```
#pragma SOMClassName(* | C++ClassName, "SOMClassName")
```

Within the scope of a DirectToSOM class, you can simply specify ***** for the first parameter, and outside you must specify the name of the C++ class to which it applies.

Some pragmas, such as **SOMNoMangling**, support both conventions:

```
#pragma SOMNoMangling( on | off | pop | * )
```

You can specify that the pragma should apply on a range basis, using **on**, **off**, or **pop**, or that it should apply just to the current class in scope, with *****. The default setting for each pragma, if applicable, is underlined in the following sections.

SOM

Syntax: **#pragma SOM**

The **SOM** pragma is used to indicate to the compiler that the class **SOMObject** should be treated as a DirectToSOM C++ class for DirectToSOM enablement. You will never need to specify this pragma—it is specified in the **<somh.hh>** header, which is included by **<som.hh>**. So once you include the **<som.hh>** header, **SOMObject** will be treated as the DirectToSOM base class.

The reason that the DirectToSOM C++ compiler doesn't just assume that **SOMObject** is a DirectToSOM class is to allow programs created using the generic C++ language bindings, which also inherit from **SOMObject**, to be compiled by the compiler. This is why you cannot mix DirectToSOM and C++ language bindings header files: the **SOMObject** class is used by both, but must be interpreted differently by the compiler.

SOMAsDefault

Syntax: `#pragma SOMAsDefault(on, off, pop)`

Specifying **SOMAsDefault(on)** indicates that the compiler should implicitly insert **SOMObject** as a base class for all subsequent classes, making them **DirectToSOM** classes. This applies to all subsequent classes declared at file scope, including classes nested within classes defined at file scope. It does not apply to any classes declared within a function block. (Recall from chapter 2 that **DirectToSOM** classes cannot be declared at block scope.) **SOMAsDefault(on)** will include the file `<som.hh>` if it has not already been included. Defining a **DirectToSOM** class using the **SOMAsDefault** pragma is known as *implicit* or *transparent* mode, whereas defining a **DirectToSOM** class by inheriting from another **DirectToSOM** class (including **SOMObject**) is known as *explicit* mode. For example, in the following code segment, class **A** is a **DirectToSOM** class, while class **B** is a native C++ class.

```
#pragma SOMAsDefault (on)

class A {
    int a;
};

#pragma SOMAsDefault (pop)

class B {
    int b;
};
```

SOMAttribute

Syntax: `#pragma SOMAttribute(DataMember, keyword1, keyword2, . . .)`

The **SOMAttribute** pragma is used to make a nonstatic data member into a CORBA attribute. In its simplest form, specifying just the **DataMember** name, the compiler implicitly generates two methods corresponding to that data member: `_set_name` and `_get_name`. The data member itself becomes private, and the `_get/_set` methods are implicitly called to access the data member in all contexts that don't have access to private class data. Making a data member into an attribute is necessary only if you want to allow other languages, such as Smalltalk, to access that data member, or if you will be using **DSOM** to access remote objects. (See Chapter 8 for additional considerations with respect to defining attributes in a **DSOM** environment.)

The **DirectToSOM** attribute support allows a data member to be accessed syntactically in C++ exactly the same way as if it were just a normal data member, but the `_get/_set` will be implicitly used if necessary.

The **SOMAttribute** pragma can be applied only to nonstatic class data members. As an example, the data member `i` in class `A` is made into an attribute at line 7 in the following program. The compiler will implicitly generate a `_get_i` method that returns the value of the data member, and a `_set_i` method that sets its value. The compiler will also make the actual data member itself (known as the *backing data*) private, so the backing data will be directly accessible only from code that has access to private class data. For any code that does not have private class access, the compiler will implicitly call the `_get_i` and `_set_i` methods to access the data member.

At line 14, access of the member `i` is in a context that has access to private class data, so the data for `i` will be accessed directly when retrieving its value to display to standard output. However, code in the `main` function does not have access to private class data, so at line 22, the compiler will implicitly invoke the `_set_i` method to assign a new value to `i`, and will invoke `_get_i` to retrieve the value of `i` at line 25.

Simple Attribute Example (*attrsimp.cpp*):

```

1  #include <iostream.h>
2  #include <som.hh>
3
4  class A : public SOMObject {
5  public:
6      int i;
7      #pragma SOMAttribute(i)
8      void display();
9  };
10
11 void A::display()
12 {
13     // uses i directly
14     cout << __FUNCTION__ "i : " << i << endl;
15 }
16
17 int main(int argc, char *argv[])
18 {
19     A a;
20
21     // uses _set_i
22     a.i = 6;
23
24     // uses _get_i
25     cout << __FUNCTION__ "i : " << a.i << endl;
26
27     a.display();
28 }
```

There are a variety of keywords that you can specify with the **SOMAttribute** pragma to effect how the compiler generates the `_get/_set`

methods and the backing data. These are (roughly in order of most common usage):

privatedata	The backing data is given private access. This is the default.
protecteddata	The backing data is given protected access.
publicdata	The backing data is given public access.
noget	The compiler does not generate a <code>_get</code> method for the attribute. Instead, you must supply the method definition yourself.
noset	The compiler does not generate a <code>_set</code> method. Instead, you must supply the method definition yourself.
readonly	The attribute does not have a <code>_set</code> method defined for it. This prevents the data member from being updated unless the backing data is accessible.
nodata	The compiler does not generate backing data or <code>_get/_set</code> methods for the attribute. You must supply both methods explicitly.
virtualaccessors	The <code>_get/_set</code> methods are defined as virtual functions.
indirect	The <code>_get/_set</code> methods are defined with an extra level of indirection.

privatedata, protecteddata, and publicdata

These keywords affect the access of the backing data, and determine whether the compiler accesses the backing data directly or uses the `_get/_set` methods. You would typically use these keywords for performance reasons, to allow the backing data to be accessed directly rather than through the `_get/_set` methods. For example, you may want to make a data member into an attribute so that it can be accessed from Smalltalk, but you don't need to have the compiler use `_get/_set` from within DirectToSOM C++. You cannot make the backing data more accessible than the attribute itself; in other words, if the attribute is protected, the backing data must be either protected or private; it cannot be public.

Note that if the object is a remote DSOM object, then accessing the backing data directly is invalid, so these keywords are not applicable to all situations. DSOM considerations for defining attributes will be discussed in detail in Chapter 8.

As an example, I have modified the previous program to specify **publicdata** for the attribute. In this case, the backing data will be public, and the `_get/_set` methods will not be used to access the data from the `main` function. The `_get/_set` methods will still be available, however, for use through other languages, for example.

Using the *publicdata* Keyword (*pubdata.cpp*):

```

1  #include <iostream.h>
2  #include <som.hh>
3
4  class A : public SOMObject {
5  public:
6      int i;
7      #pragma SOMAttribute(i, publicdata)
8      void display();
9  };
10
11 void A::display()
12 {
13     // uses i directly
14     cout << __FUNCTION__ "i : " << i << endl;
15 }
16
17 int main(int argc, char *argv[])
18 {
19     A a;
20
21     // uses i directly
22     a.i = 6;
23
24     // uses i directly
25     cout << __FUNCTION__ "i : " << a.i << endl;
26
27     a.display();
28 }

```

noget and noset

The names **noget** and **noset** are often a point of confusion. They do not imply that a **_get** or **_set** method is unavailable for the attribute, only that the compiler does not generate it and you must supply it. In many cases, the compiler-supplied attribute methods are sufficient. But there are situations, as we shall see in Chapter 8, where you may need to modify these methods. In addition, specifying your own attribute methods allows you to perform additional work whenever a data member is accessed.

To illustrate, I have modified the first attribute example to supply user-defined **_get** and **_set** methods as shown in the program example that follows. As with the first example, the **_get/_set** methods will be called for the data access in the main function, and the data will be accessed directly from within the class implementation. Note that with VisualAge C++ for OS/2, attribute methods were originally defined with volatile qualifiers and volatile parameters by default. In CSD 2 for OS/2, the **-yxqnosomvolattr** compiler option was added to indicate that the attributes should not be volatile-qualified. This is the default for Windows and will be the default for the next

release of the OS/2 product. Specifying this option indicates that the attribute methods should not be volatile-qualified. The actual option is **qsomvolattr/qnosomvolattr** in the Windows release and for the next release of the OS/2 product. The **yx** was needed in front to add an option in a CSD. In Windows, you can specify the **-qsomvolattr** flag to indicate that attributes should be volatile-qualified for upward compatibility. I recommend that you not use volatile-qualified attributes at all, and stick with the default. The example program was compiled in OS/2 with the following command:

```
icc -yxqnosomvolattr attr3.cpp
```

I typically set the OS/2 **ICC** environment variable to be **-yxqnosomvolattr**. The output for the program is:

```
A::_set_i(int)
A::_get_i() const
main(int,char**)i : 6
A::display() i : 6
```

Using noget/noset (noget.cpp):

```
1  #include <iostream.h>
2  #include <som.hh>
3
4  class A : public SOMObject {
5      public:
6          int i;
7          #pragma SOMAttribute(i, noget, noset)
8          void display();
9  };
10
11 void A::display()
12 {
13     // uses i directly
14     cout << __FUNCTION__ " i : " << i << endl;
15 }
16
17 int A::_get_i() const
18 {
19     cout << __FUNCTION__ << endl;
20     return i;
21 }
22
23 void A::_set_i(int j)
24 {
25     cout << __FUNCTION__ << endl;
26     i = j;
27 }
28
29 int main(int argc, char *argv[])
```

```

30 {
31     A a;
32
33     // uses _set_i
34     a.i = 6;
35
36     // uses _get_i
37     cout << __FUNCTION__ "i : " << a.i << endl;
38
39     a.display();
40 }

```

The **_get** and **_set** methods have different signatures depending upon the underlying data type. The **_get** method is always a **const** method, as it should never modify the data value. For address-based parameters, the **_set** method parameter is defined as **const**, because the input value should not be modified either. The following shows the method signatures for each data type. Note that I have intentionally placed the type, **T**, before the **const** qualifier, because for types such as pointers, **const T** is not equivalent to **T const**. The former is not correct for the attribute signature, while the latter is.

- ♦ Scalar of type T:

```

T _get_var() const;
void _set_var(T);

```

- ♦ Scalar of type T with **indirect** keyword (**indirect** affects only the signatures of scalar attributes):

```

&T _get_var() const;
void _set_var(T const &);

```

- ♦ Arrays of elements of type T:

```

T* _get_var() const;
void _set_var(T const *);

```

- ♦ Native C++ classes of type T:

```

T _get_var() const;
void _set_var(T const &);

```

- ♦ SOM classes of type T:

```

T* _get_var() const;
void _set_var(T const *);

```

And here's an example showing each signature:

Attribute Method Signatures (getset.cpp):

```

1  #include <som.hh>
2
3  class A {
4      int i;
5  };
6
7  class B : public SOMObject {
8      int i;
9  };
10
11 class C : public SOMObject {
12 public:
13
14     #pragma SOMdefine(*)
15
16     int f();
17
18     // Direct scalar attribute:
19     int scalar;
20     #pragma SOMAttribute(scalar, noget, noget)
21
22     // Indirect scalar attribute:
23     int scalarIndirect;
24     #pragma SOMAttribute(scalarIndirect, noget, noget, indirect)
25
26     // Direct scalar pointer attribute:
27     char *ptrDirect;
28     #pragma SOMAttribute(ptrDirect, noget, noget)
29
30     // Indirect scalar pointer attribute:
31     char *ptrIndirect;
32     #pragma SOMAttribute(ptrIndirect, noget, noget, indirect)
33
34     // Array attribute:
35     char array[255];
36     #pragma SOMAttribute(array, noget, noget)
37
38     // Class attribute:
39     A classInstance;
40     #pragma SOMAttribute(classInstance, noget, noget)
41
42     // SOM class attribute:
43     B somClassInstance;
44     #pragma SOMAttribute(somClassInstance, noget, noget)
45 };
46
47 int C::f()
48 {
49     return 6;
50 }
51
52 void C::_set_scalar(int newValue)

```

```

53 {
54     scalar = newValue;
55 }
56
57 int C::_get_scalar() const
58 {
59     return scalar;
60 }
61
62 void C::_set_scalarIndirect (int const & newValue)
63 {
64     scalarIndirect = newValue;
65 }
66
67 int & C::_get_scalarIndirect() const
68 {
69     // cast away const of this pointer
70     return (int &)scalarIndirect;
71 }
72
73 void C::_set_ptrDirect(char * newstr)
74 {
75     ptrDirect = newstr;
76 }
77
78 char * C::_get_ptrDirect() const
79 {
80     return ptrDirect;
81 }
82
83 void C::_set_ptrIndirect(char * const & newstr)
84 {
85     ptrIndirect = newstr;
86 }
87
88 char *& C::_get_ptrIndirect() const
89 {
90     // cast away const of this pointer
91     return (char *&)ptrIndirect;
92 }
93
94 void C::_set_array(const char * new_array)
95 {
96     strcpy((char *)array, (const char *)new_array);
97 }
98
99 char *C::_get_array() const
100 {
101     // cast away const of this pointer
102     return (char *)array;
103 }
104
105 void C::_set_classInstance(const A & aref)
106 {

```

Continued

```

107     classInstance = aref;
108 }
109
110 A C::_get_classInstance() const
111 {
112     // cast away const of this pointer
113     return (A)classInstance;
114 }
115
116 void C::_set_somClassInstance(const B * bref)
117 {
118     somClassInstance = *bref;
119 }
120
121 B* C::_get_somClassInstance() const
122 {
123     // cast away const of this pointer
124     return &((C *)this)->somClassInstance;
125 }

```

readonly

When **readonly** is specified for an attribute, a **_set** method will not be made available. Specifying the **readonly** keyword essentially makes an attribute into a constant for any code that does not have access to the backing data. From the client perspective, designating a **const** data member as an attribute is equivalent to specifying **readonly**, as neither will have a **_set** method available.

For example, in the following program, the assignment to *i* at line 29 in the **main** function is now invalid because the backing data is not accessible and because the attribute is designated as **readonly**, so a **_set** method is not available either. However, the data member can still be updated in the class implementation, as shown at line 15 in the method **set**.

Using readonly (readonly.cpp):

```

1  #include <iostream.h>
2  #include <som.hh>
3
4  class A : public SOMObject {
5  public:
6      int i;
7      #pragma SOMAttribute(i, readonly)
8      void display();
9      void set(int);
10 };
11
12 void A::set(int j)
13 {
14     // uses i directly
15     i = j;
16 }

```



```

17
18 void A::display()
19 {
20     // uses i directly
21     cout << __FUNCTION__ "i : " << i << endl;
22 }
23
24 int main(int argc, char *argv[])
25 {
26     A a;
27
28     // invalid: no _set_i
29     a.i = 6;
30
31     a.set(4);
32
33     // uses _get_i
34     cout << __FUNCTION__ "i : " << a.i << endl;
35
36     a.display();
37 }

```

nodata

When ***nodata*** is specified for an attribute, the compiler does not generate backing data or ***_get/_set*** methods for the attribute. You must supply both methods explicitly. Even though the data member is declared in the class, no instance data storage is allocated for it in a created object. The compiler therefore cannot generate default ***_get*** and ***_set*** methods, because there is no data member to access. Any attempts to access the instance data member by name, even from within the class implementation, will invoke the corresponding ***_get*** or ***_set*** method. Note that because there is no instance data allocated for such attributes, you cannot take their addresses.

For example, in the following program, the data member ***i*** is declared as an attribute with ***nodata***. This implies that storage will not be allocated for ***i*** as part of the instance data for classes of type ***A***. In this simple example, I've defined a static member ***lastSetting*** that is used to hold the last value specified with a ***_set_i***; ***_get_i*** which simply returns this value. Note that because there is no data member, the access of ***i*** through the display method must also invoke the ***_get_i*** method. The output for the program is:

```

sizeof(A): 4
A::_set_i(int)
A::_get_i() const
main(int,char**)i : 6
A::_get_i() const
A::display() i : 6

```

As we shall see in Chapter 6, every SOM object contains a pointer in the beginning that provides addressability to the SOM class data struc-

tures. A size of 4 in this example indicates that the instance size contains room only for this pointer; no instance data will be allocated as part of the object.

Using *nodata* (*nodata.cpp*):

```

1  #include <iostream.h>
2  #include <som.hh>
3
4  class A : public SOMObject {
5      static int lastSetting;
6      public:
7          int i;
8          #pragma SOMAttribute(i, nodata)
9          void display();
10 };
11
12 int A::lastSetting 0;
13
14 void A::display()
15 {
16     // calls _get_i
17     cout << __FUNCTION__ " i : " << i << endl;
18 }
19
20 int A::_get_i() const
21 {
22     cout << __FUNCTION__ << endl;
23     return lastSetting;
24 }
25
26 void A::_set_i(int j)
27 {
28     cout << __FUNCTION__ << endl;
29     lastSetting = j;
30 }
31
32 int main(int argc, char *argv[])
33 {
34     A a;
35
36     cout << "sizeof(A): " << sizeof(A) << endl;
37     // uses _set_i
38     a.i = 6;
39
40     // uses _get_i
41     cout << __FUNCTION__ " i : " << a.i << endl;
42
43     a.display();
44 }

```

virtualaccessors

The **virtualaccessors** keyword causes the compiler to define the `_get/_set` methods as virtual functions. This modifies the native C++ data member access rules. Normally, a data member is not overridden in a derived class. But by defining virtual attribute methods, a data member of the same name as one in a base class will override that base class data member. Unfortunately, I was unable to create a working example using this keyword, due to a compiler problem that was common across all platforms. This has been reported, and should be fixed in either an upcoming CSD or the next release of the products.

indirect

When the `indirect` keyword is specified, the `_get/_set` methods are defined with an extra level of indirection. This is applicable only to scalar attributes—it is ignored for arrays and classes. The intention is to allow a scalar attribute to be implicitly passed by reference, instead of by value. It would typically be used in a DSOM environment, but is not necessary for DSOM.

The following example shows how to use the **indirect** keyword. The only difference between this example and the **noget/noset** example shown previously is that the attribute methods accept and return references; the compiler will implicitly pass and accept references when the attributes are used.

Using indirect (indirect.cpp):

```

1  #include <iostream.h>
2  #include <som.hh>
3
4  class A : public SOMObject {
5  public:
6      int i;
7      #pragma SOMAttribute(i, indirect, noget, noset)
8      void display();
9  };
10
11 void A::display()
12 {
13     // uses i directly
14     cout << __FUNCTION__ " i : " << i << endl;
15 }
16
17 int &A::_get_i() const
18 {
19     // cast away const of this
20     return ((A *)this)->i;
21 }
22
```

Continued

```

23 void A::_set_i(int const &j)
24 {
25     i = j;
26 }
27
28 int main(int argc, char *argv[])
29 {
30     A a;
31
32     // uses _set_i
33     a.i = 6;
34
35     // uses _get_i
36     cout << __FUNCTION__ "i : " << a.i << endl;
37
38     a.display();
39 }

```

SOMCallStyle

Syntax: `#pragma SOMCallStyle(IDL | OIDL)`

The CORBA architecture requires that all methods accept a second argument that is of type **Environment** (the first argument being the address of the target object itself). Earlier versions of SOM did not support the **Environment** parameter, so methods of classes defined using these earlier versions do not accept an **Environment** parameter. Whether or not a method accepts the **Environment** parameter is referred to as the *callstyle* for that method. Methods that expect the **Environment** parameter are classified as callstyle **IDL**, whereas those that don't are callstyle **OIDL** (for Object IDL—the name of the original SOM IDL language prior to the adoption of CORBA IDL). The callstyle is specified at the class level, so all methods introduced in a class either have callstyle **IDL** or **OIDL**. Note that this applies only to methods introduced by the class. The callstyle of overridden virtual functions will be that of the introducing class.

By default, all DirectToSOM classes that you define will have callstyle **IDL**. This can be modified with the **SOMCallStyle** pragma, but you probably won't ever need to. See Chapter 5, in the *Environment Parameter and Error Handling* section for information about using the **Environment** structure.

SOMClassInit

Syntax: `#pragma SOMClassInit(* | C++ClassName, C++Prototype)`

The **SOMClassInit** pragma specifies a function to be called after the class object has been created. This can be used by the class implementation to

perform specific actions when the class object is first created. The specified function must accept a single parameter of type **SOMClass***, which will be a pointer to the class object that was just created. Note that this function must be defined with the appropriate linkage for SOM functions. The SOM header files define two macros **SOMEXTERN** and **SOMLINK**, that you can use to specify the linkage in a platform-independent way.

In the next programming example, the function `classInit` is specified as the class initialization function for both classes A and B. This function will be called after the class objects for either of these classes are created, and will invoke the **somGetName** method against the class object to display the name of the class object that was created. The use of the **SOMDefine** pragma is discussed later in this chapter. The output for this program is:

```
classInit(SOMClass*) called for class za
classInit(SOMClass*) called for class zb
```

Note that the class names have been mangled to case-insensitive lowercase names. This mapping is discussed in more detail in Chapter 6, *Inside DirectToSOM C++*.

Using **SOMClassInit** (*clsinit.cpp*):

```
1  #include <iostream.h>
2  #include <som.hh>
3
4  SOMEXTERN void SOMLINK classinit(SOMClass *cls)
5  {
6      cout << __FUNCTION__
7           << " called for class "
8           << cls->somGetName() << endl;
9  }
10
11 class A : public SOMObject {
12     #pragma SOMClassInit(*, classinit(SOMClass *))
13     #pragma SOMDefine(*)
14 };
15
16 class B : public SOMObject {
17     #pragma SOMClassInit(*, classinit(SOMClass *))
18     #pragma SOMDefine(*)
19 };
20
21 int main(int argc, char *argv[])
22 {
23     A a;
24     B b;
25 }
```

SOMClassName

Syntax: `#pragma SOMClassName(* | C++ClassName, "SOMClassName")`

The **SOMClassName** pragma defines the SOM name for a class. By default, C++ class names are mangled to form case-insensitive names, because SOM is not case-sensitive. This mapping is discussed in more detail in Chapter 6, *Inside DirectToSOM C++*, but the most commonly encountered mangling is that of translating uppercase letters to their lowercase counterpart prefixed with a z. For example A becomes za.

You can use the **SOMClassName** pragma to assign an unmangled name or any name that you choose, for the class, but you cannot assign the same name to two different classes. Unless you are working in a mixed-language environment, this pragma is typically not necessary. The use of this pragma is discussed in more detail in Chapters 7 and 9.

As an example, I've modified the class definitions for the previous example of the **SOMClassInit** pragma to specify the **SOMClassName** pragma for each class. The output for the program is now:

```
classInit(SOMClass*) called for class A
classInit(SOMClass*) called for class B
```

Using *SOMClassName(clsname.cpp)*:

```

1  #include <iostream.h>
2  #include <som.hh>
3
4  SOMEXTERN void SOMLINK classinit(SOMClass *cls)
5  {
6      cout << __FUNCTION__
7          << " called for class "
8          << cls->somGetName() << endl;
9  }
10
11 class A : public SOMObject {
12     #pragma SOMClassInit(*, classinit(SOMClass *))
13     #pragma SOMClassName(*, "A")
14     #pragma SOMDefine(*)
15 };
16
17 class B : public SOMObject {
18     #pragma SOMClassInit(*, classinit(SOMClass *))
19     #pragma SOMClassName(*, "B")
20     #pragma SOMDefine(*)
21 };
22
23 int main(int argc, char *argv[])
24 {
25     A a;
26     B b;
27 }
```

SOMClassVersion

Syntax: `#pragma SOMClassVersion(* | C++ClassName, MajorVersion, MinorVersion)`

When a client first attempts to use a SOM class object, it implicitly supplies version information that SOM will check to ensure that the use of the class and its implementation are compatible. When this pragma is supplied, the version information specified is stored with both the class implementation and the client. When the client program first attempts to use that class, it will pass the specified version information. If the **SOMClassVersion** pragma is not supplied, the default is 0 for both the major and minor versions.

When a client first uses a SOM class, SOM will check that the major versions requested by the client and supported by the implementation are the same. If they are not, the client and the implementation are assumed to be incompatible, and an error occurs. If the minor version requested by the client is higher than that supported by the implementation, a warning will be issued indicating that a back-level implementation is being used with the client, which could indicate possible incompatibilities. This checking occurs only the first time an object of that class is instantiated. Incompatibilities may therefore go undetected if several different objects modules are compiled into a single client, where the object modules were compiled with different values of the **SOMClassVersion** pragma.

The following shows an example of how this pragma is used. Note that it is very much a contrived example, as you would typically specify the **SOMClassVersion** pragma only once, with the class definition itself, and not separately with the client and the implementation. However, this example makes it relatively easy to illustrate the concepts. The implementation is compiled with a class version of (1,0), but the client is compiled with a class version of (2,0). When this program is run, an error such as the following will be generated:

```
Error: class <zversion> version mismatch (version compatible with (2,0) requested
by caller, but version (1,0) is provided by the class's implementation)
Error: Attempt to load, create or use an incompatible class.
```

Definition of Class Version (version.hh):

```
1 #include <som.hh>
2
3 class Version : public SOMObject {
4 };
```

Implementation of Class Version (versiona.cpp):

```
1 #include "version.hh"
2
```

Continued

```

3 #pragma SOMClassVersion(Version, 1, 0)
4
5 #pragma SOMDefine(Version)

```

Client of Class Version (versionb.cpp):

```

1 #include "version.hh"
2
3 #pragma SOMClassVersion(Version, 2, 0)
4
5 int main(int argc, char *argv[])
6 {
7     Version v;
8 }

```

SOMDataName

Syntax: `#pragma SOMDataName(C++DataMemberName, "SOMName")`

The **SOMDataName** pragma defines the SOM name for the data member of a class. By default, data member names are mangled to form case-insensitive names, because SOM is not case-sensitive (this mapping is discussed in more detail in Chapter 6, *Inside DirectToSOM C++*). This pragma allows you to supply the SOM name that should be used for a data member, rather than the mangled name. Note that you can also use the **SOMNoMangling** pragma to prevent member name mangling. The **SOMDataName** pragma is necessary only if you want to supply a completely different name, or if two data member names differ only by case; otherwise, **SOMNoMangling** will suffice; further, preventing mangling of data member names is necessary only if the class will be used with languages other than *DirectToSOM C++*. See Chapter 7, *IDL Generation*, the *Name Mangling*, section for further discussion of the **SOMDataName** pragma.

SOMDefine

Syntax: `#pragma SOMDefine(* | on | off | pop | C++ClassName)`

The **SOMDefine** pragma is necessary for generating the SOM class data structures for classes that have no out-of-line member functions. This pragma is ignored if the class has at least one out-of-line data member.

For each SOM class implementation, the *DirectToSOM C++* compiler must generate several data structures and export three symbols for use by the SOM run time. The exported symbols are `<class>ClassData`, `<class>CClassData`, and `<class>NewClass`. (The meaning and use of these symbols will be explained in detail in Chapter 6, *Inside DirectToSOM C++*). But the compiler should generate these structures

and symbol exports only once per class implementation, otherwise there would be wasted storage and possible duplicate declaration problems.

For classes that have at least one out-of-line function, the implementation is defined as the file where the definition of the first nonstatic out-of-line member function is defined. The compiler generates the SOM class data structures and symbol exports as part of compiling this file. This ensures that the class data structures are defined only once for that class implementation. In the following example, the SOM class data structures for class A will be generated with the file that contains the definition for the member function show.

```

1  #include <som.hh>
2
3  class A : public SOMObject {
4  private:
5      int i;
6  public:
7      A() { i = 0; }
8      void show();
9      void set i(int newvalue) { i = newvalue; }
10     int get i() { return i; }
11 };

```

For classes that have all inline or no member functions, there is no way for the compiler to determine where to generate the structures. In such cases, you must explicitly indicate where the structures should be generated using the **SOMDefine** pragma. For example, the class A shown next has no out-of-line member functions. Therefore, the compiler will not generate the SOM class data structures for A until it encounters a **SOMDefine** pragma for A. If the file shown here were compiled into a program, without the **SOMDefine** pragma at line 9, it would generate unresolved symbol errors at link time because the SOM class symbols would not be generated for the class A.

```

1  #include <som.hh>
2
3  #pragma SOMAsDefault(on)
4
5  class A {
6      int i;
7  public:
8      A() { i = 0; };
9      #pragma SOMDefine(*)
10 };
11
12 int main(int argc, char *argv[])
13 {
14     A a;
15 };
16

```

The compiler will generate the SOM class data structures and symbol exports each time it encounters the **SOMDefine** pragma for a given class. It is therefore not a good idea to include the pragma with the class definition, but rather, in a separate file. Otherwise the compiler will generate the structures each time the header file is parsed, resulting in wasted storage and possible duplicate definition link errors.

SOMIDLDecl

Syntax: `#pragma SOMIDLDecl(C++TypeName | C++Prototype, "IDL Declaration")`

The **SOMIDLDecl** pragma is used to modify the IDL declaration for a given type or class member. A typical use of this pragma is to modify the parameter directional attributes for address-based parameters. See Chapter 7, the *Modifying Generated IDL Declarations* section, for examples of using this pragma.

SOMIDLPass

Syntax: `#pragma SOMIDLPass(* | C++ClassName, "Label", "IDLString")`

The **SOMIDLPass** pragma is used to generate an arbitrary IDL string to the IDL file with the interface for the given class. The *Label* parameter indicates where within the interface declaration the string will be generated, with the following meanings:

Begin	At the beginning of the IDL file, following the #ifdef and #define directives that guard inclusion of the file.
End	At the end of the file, prior to the penultimate #endif .
Interface-Begin	Directly after the opening brace for the class interface definition.
Interface-End	Directly before the closing brace for the class interface definition.
Implementation-Begin	Directly after the opening brace for the class implementation section.
Implementation-End	Directly before the closing brace for the class implementation section.

If a label other than one of these is specified, the pragma will be ignored, but no warning will be issued. You can specify multiple **SOMIDLPass** pragmas for a given class; they are cumulative. See Chapter 7, the *General Modifiers* section, for examples of using the **SOMIDLDecl** pragma.

SOMIDLTypes

Syntax: `#pragma SOMIDLTypes(* | C++ClassName, typename1, typename2, ...)`

The **SOMIDLTypes** pragma is used to force the generation of specific types into the IDL file along with the interface definition for the given class. By default, the compiler generates corresponding IDL declarations only for types that are defined in the current file. If a type appears in a separate file, it will not be generated in the IDL for the current file, the assumption being that the type will be included separately. For each **.hh** file included in the current file, a corresponding **.idl** file will be included in the generated **.idl** file, which will handle most type dependencies. But if the type appears in an **.h** file for example, a corresponding include directive will not be generated in the **.idl** file. If that **.h** file contains a type that the generated IDL depends upon, you may need to have that type definition generated directly into the **.idl** file. For such situations, you can use the **SOMIDLTypes** pragma.

In the following example, class **A** depends upon the type **myType**, which is defined in the file **idltype2.h**. Because **idltype.h** is not an **.hh** file, a corresponding include of an **.idl** file will not be generated in the **.idl** file for class **A**. In order to resolve the dependency upon **myType**, the **SOMIDLTypes** pragma is used to force the compiler to generate **myType** into the **.idl** file.

Type definition for myType (file idltype2.h):

```
1 typedef int myType;
```

DirectToSOM C++ Class A (file idltype.hh):

```
1 #include <som.hh>
2
3 #include "idltype2.h"
4
5 class A : public SOMObject {
6     #pragma SOMIDLTypes(*, myType)
7     myType m;
8 };
```

You can specify multiple **SOMIDLTypes** pragmas for a given class; they are cumulative, and the specified types will be generated into the IDL file in the order of occurrence of the pragmas.

SOMMetaClass

Syntax: **#pragma SOMMetaClass**(* | C++ClassName, * | "SOMClassName" | C++MetaClassName)

The **SOMMetaClass** pragma is used to specify the metaclass for the given class. Unlike C++, a SOM class exists at run time as a SOM object. Because they are objects, SOM class objects have to be instances of some class. By default, they are instances of the *metaclass* **SOMClass**. A metaclass introduces the methods and instance data that are supported by class objects. For example, **SOMClass** introduces the method **somNew**, which can be invoked against a class object to create and return a new instance of that class.

Metaclasses are used to manage and control the creation of SOM objects. You can define your own metaclass for a class object, for example, to limit the number of instances created to five. SOM also supplies several metaclasses with the Metaclass Framework. Typically, though, **SOMClass** will suffice for most applications.

One use of metaclasses is to provide data that is shared across all instances of a class. There will be only one instance of the metaclass created, no matter how many instances of that class are created. You can access the metaclass object for a SOM object and use the metaclass to implement the equivalent of static data members in C++. Because static data members cannot be made into attributes, defining a metaclass with an attribute allows data to be shared across languages by all instances of a class.

The following program illustrates this process. The class **A** inherits from the class **SOMClass** (all metaclasses must inherit ultimately from **SOMClass**). It contains the definition for the attribute **i**. Class **B** is defined with **A** as its metaclass at line 19. When the class object for **B** is created, it is as an instance of class **A**, rather than as an instance of **SOMClass**. At line 24, an instance of **B** is created. Then at line 26, the class object for the **b** instance is accessed through the **_ClassObject** operator (this operator will be discussed in more detail in Chapter 5), and the attribute **i** is updated and displayed.

Using the SOMMetaClass pragma (meta.cpp):

```

1  #include <iostream.h>
2  #include <som.hh>
3  #include <sombacl.s.hh>
4
5  class A : public SOMClass {
6      public:
7          int i;
```

```

8   #pragma SOMAttribute(i, noset)
9   #pragma SOMDefine (*)
10  };
11
12  void A::_set i(int j) {
13      cout << __FUNCTION__ << endl;
14      i = j;
15  }
16
17  class B : public SOMObject {
18      #pragma SOMMetaClass(*, A)
19      #pragma SOMDefine (*)
20  };
21
22  int main(void)
23  {
24      B b;
25
26      {(A *)b. ClassObject)->i 10;
27      cout << "i is "
28              << {(A *)b. ClassObject)->i << endl;
29  }

```

SOMMethodName

Syntax: `#pragma SOMMethodName(C++MemberFunctionName, "SOM-Name")`

The **SOMMethodName** pragma defines the SOM name for the data member of a class. By default, SOM class member function names are mangled to include parameter type information and then to form case-insensitive names, because SOM is not case-sensitive (this mapping is discussed in more detail in Chapter 6, *Inside DirectToSOM C++*). This pragma allows you to supply the SOM name that should be used for a member function, rather than the mangled name. You can also use the **SOMNoMangling** pragma to prevent member name mangling. The **SOMMethodName** pragma is necessary only if you want to supply a completely different name, or if two data member names differ only by case; otherwise, **SOMNoMangling** will suffice. Further, preventing mangling of data member names is necessary only if the class will be used with languages other than DirectToSOM C++. See Chapter 7, *IDL Generation*, the *Name Mangling* section, for further discussion and an example of using the **SOMMethodName** pragma.

SOMModule

Current Syntax: `#pragma SOMModule(C++ClassName)`

The **SOMModule** pragma is currently generated into **.hh** files by the SOM compiler in the SOMObjects 3.0 beta. It provides a mapping from the IDL

module statement. However, no DirectToSOM C++ compiler currently supports this pragma.

Multiple interface definitions in the same **.idl** file can be grouped together by enclosing the definitions inside a **module** statement. The SOM names of any classes inside a module become the class name prefixed with the module name. The **hh** emitter will generate a **SOMModule** pragma in the resulting **.hh** file to indicate the module name; in which case, the compiler must also use the combined module and class name in making references to the SOM class data structures.

Because IDL modules are used extensively in the Object Services IDL definitions, it is expected that the next releases of the compiler products will support this pragma. The current syntax generated by the SOM compiler is shown here, but it is anticipated that this will change slightly.

SOMNoDataDirect

Syntax: **#pragma SOMNoDataDirect(* | on | off | pop)**

The **SOMNoDataDirect** pragma is used in conjunction with the **SOMAttribute** pragma to further control when the **_get/_set** methods are invoked for accessing a data member. When **SOMNoDataDirect** is in effect, all data member access is performed through the **_get/_set** methods, except data accessed through the **this** pointer. This implies that all data members must be made into attributes, otherwise compile-time errors will occur when the compiler attempts to use the **_get/_set** methods. This pragma is relevant only when DSOM is being used—see Chapter 8, *Distributed SOM*, for further discussion of this pragma.

SOMNoMangling

Syntax: **#pragma SOMNoMangling(* | on | off | pop)**

SOMNoMangling turns off name mangling and case-insensitive conversion for function and data member names, so that the SOM name for each member will be the same as the C++ name. Because SOM does not support method name overloading by parameter types, if you do have overloaded method names in a DirectToSOM C++ class, specifying **SOMNoMangling** will result in compile-time errors for each overload, indicating that the SOM method name has already been used for that class. For such situations, you must use the **SOMMethodName** to explicitly provide a different name for each overloaded method. In addition, if you have data members names that differ only by case, you will need to use the **SOMDataName** pragma to assign unique names.

The **SOMNoMangling** pragma applies only to class data member names, although a future release of the compiler may extend this to apply to class names also. Specifying **SOMNoMangling** will not prevent case-insensitive conversion of class names. In order to produce an unmangled SOM name for a class, you must use the **SOMClassName** pragma, specifying the desired SOM class name. See Chapter 7, *IDL Generation*, for further discussion and examples of using the **SOMNoMangling** pragma.

SOMNonDTS

This pragma is not intended to be used directly by DirectToSOM C++ programmers. It does not indicate that the class is not handled through DirectToSOM C++, but rather that the class was not originally defined through it.

This pragma is generated by the SOM compiler into an **.hh** file to indicate that the class is not defined by DirectToSOM C++. The reason is that DirectToSOM C++ assumes certain overrides and other information about a class. If the class is not implemented in DirectToSOM C++, then these assumptions should not be made by the compiler. For example, if you define a class using IDL, then generate an **.hh** file from that IDL definition, the generated file will include this modifier indicating that the class was not defined originally in DirectToSOM C++. For further information about generating an **.hh** file from an **.idl** file, see Chapter 7, *IDL Generation*.

SOMReleaseOrder

Syntax: `#pragma SOMReleaseOrder(element1, element2, . . .)`

One of the major underpinnings of the current SOM support for RRBC is the concept of a *release order*. Every class has a release order, which defines the order in which member functions and static data members introduced by the class are “released” from that class. SOM maintains binary compatibility by assigning each member function and static data member a specific location in the release order list; clients locate these members by their position within the release order list. As long as the order remains invariant, RRBC is maintained.

The release order maps to a table exported by the class implementation called the **<className>ClassData** structure; member functions and static data members introduced by the class are accessed by clients through their release order slot in the exported table (see Chapter 6, *Inside DirectToSOM C++*, for details). In order to maintain RRBC for a class, the release order must remain invariant, with the exception that new members can be added to the end of the release order list.

By default, the release order for a class is assumed to be the order in which these members appear within the class definition. New members must be added after all existing ones in order to maintain RRBC. You can also explicitly supply the release order through the **SOMReleaseOrder** pragma, and can then add members anywhere in the class and simply add the name to the end of release order list.

Currently, you can specify only a single **SOMReleaseOrder** pragma per class, which if specified, must contain all member functions, static data members, and attributes that are declared in the class, regardless of the access of those members. Nonstatic data members do not appear in the release order. It is expected that the next release of the compiler products will allow you to specify the release order list as multiple partial **SOMReleaseOrder** pragmas, the combination of which must specify a complete release order.

The elements specified in the release order can be one of the following:

- ♦ *Asterisk (*)*: The asterisk can be used as a placeholder to reserve a slot in the release order so that you can later add an element there without breaking RRBC. You can also use it as a placeholder for private data members if you are supplying a separate client version of a class header that contains definitions only for the public class interface.
- ♦ *Static data member*: Static data members are also accessed through the release order. All static data members in the class must appear in the release order list.
- ♦ *Attribute*: Because an attribute introduces the two methods for a class, **_get** and **_set**, attributes must also appear in the release order. Unlike the other release order elements, which only take one slot in the **<className>ClassData** structure, an attribute always takes up two slots, one for each method, even if the attribute is **const** or **readonly**. You can specify an attribute element either by supplying the single data member name or both the attribute method names. I recommend specifying the data member name.
- ♦ *Member function*: Any member function introduced by the class must appear in the release order list. Virtual functions that are introduced in a base class do not appear in the release order for a class. You can specify a member function either by just its name, if the name is unambiguous in the class, or by the full member function prototype (excluding the return value), or by the SOM method name in quotes. For example, the following shows a simple class with a corresponding **SOMReleaseOrder** pragma:

```
#include <som.hh>

class A : public SOMObject {
public:
    int foo(char*);
    #pragma SOMMethodName(foo, "somFoo")
    #pragma SOMReleaseOrder(foo)
};
```


You could also specify the release order for A in either of the following ways:

```
#pragma SOMReleaseOrder(foo(char *))
#pragma SOMReleaseOrder("somFoo")
```

- ♦ *Member Function preceded by exclamation point (!)*: A member function is preceded by an ! to indicate that it was migrated up the class hierarchy. By default, a virtual function appears only in the release order and the **<className>ClassData** structure of the class that introduced it. Method access is performed by accessing the slot in the introducing class. However, if a method is migrated up the class hierarchy, existing clients will continue to use the slot in the derived class that originally introduced the member until they are recompiled. If the corresponding slot in the derived class **<className>ClassData** structure were removed, this would break RRBC. Specifying an ! causes the compiler to reserve a slot in the **<className>ClassData** data structure so that existing client can continue to use that slot. However, the member function will be considered introduced in the base class to which it was migrated. See Chapter 3, *Release-to-Release Binary Compatibility*, for further discussion and examples of using the !.

The release order list should contain only static data members, attributes, and member functions that are introduced by the class, regardless of access. Data members that are not attributes do not appear in the release order. Destructors and certain constructors become overrides of the 10 special **SOMObject** methods, so they do not appear in the release order. Only nonvirtual functions and newly introduced virtual functions appear in the release order for a class. You can use the **/Fr** option under OS/2 or Windows to have the compiler generate the release order for you, which you can then insert into the class. See the discussion of compiler options forthcoming.

The four **SOMObject** assignment methods also do not have to appear in the release order, even if you override them explicitly in the class, because they are introduced by **SOMObject**. If you supply an **operator=**, you will need to put this method into the release order, as **operator=** does not override the SOM assignment methods. (See Chapter 6, *Inside DirectToSOM C++*, for further details.)

By default, the compiler-supplied **operator=** methods, if any, are not part of the public interface to the class and do not appear in the release order. You can put them in the release order, however, which will make them part of the interface. This is typically necessary only if you want to take the address of that member function. For example, in the following class, the compiler-defined **operator=** method is specified in the release order, making it part of the public class interface. You still do not need to supply the body, and this does not affect the default method generation performed by the compiler.

```
#include <som.hh>

class A : public SOMObject {
public:
    #pragma SOMReleaseOrder(operator=(const A&))
};
```

Macros Defined for DirectToSOM

There is currently one compiler-defined macro generated for DirectToSOM: **__SOM_ENABLED__**. This macro is defined with a positive integer value representing the version of SOM supported by the current compiler. If this macro is not defined, or is defined with a value of 0, then DirectToSOM support is not provided by the current compiler. For the OS/2 and Windows compilers that I used in writing this book, the value of this macro is 210, representing SOMObjects version 2.1.

Compiler Options

The following compiler options are currently supported by the VisualAge C++ products for OS/2 and for Windows. The DirectToSOM compilers on other platforms also support these options, but the syntax will be different.

- /Fr** Generates release order.
- /Fs** Generates IDL file.
- /Ga** Enables implicit SOM mode processing.
- /Gb** Disables direct data access for data members.
- /Gz** Performs class initialization only as needed at run time.
- /Xs** Excludes files from implicit SOM mode processing.

/Fr

Syntax: /Fr<className>

The **/Fr** option instructs the compiler to generate the release order for the specified class to standard output. You can use this option to generate an initial release order for a class if you are just creating it; or you can generate the existing compiler-defined release order and insert this in the class header. This will allow you to add new members anywhere within that existing class

without breaking RRBC or being forced to add the member to the end of the class; you can simply append them to the end of the release order list.

For example, if the file containing the following class were compiled with `icc /Fr A:`

```
class A : public SOMObject {
public:
    void foo();
    void foo(int);
    void bar(int);
    void bar(char *);
};
```

the following would be written to standard output:

```
/* A */
#pragma SOMReleaseOrder( \
/* 1 */    foo(),\
/* 2 */    foo(int),\
/* 3 */    bar(int),\
/* 4 */    bar(char*))
```

/Fs

Syntax: `/Fs [± | - filename | directory]`

The **/Fs** option is used to control IDL generation for DirectToSOM C++ class definitions. The default is **/Fs+**, which means to generate a corresponding **.idl** file for every **.hh** file that is specified on the command line. See Chapter 7, *IDL Generation*, for a detailed discussion of this option and its use.

/Ga

Syntax: `/Ga[+ | =]`

The **/Ga** option enables implicit SOM mode processing by inserting the **#pragma SOMAsDefault(on)** at the beginning of each compilation unit. This will cause all subsequent classes declared at file scope or nested within classes declared at file scope to implicitly inherit from **SOMObject**. It does not apply to any classes declared within a function block. The header file **<som.hh>** will be included also. The default is **/Ga-**.

Defining a DirectToSOM class using the **SOMAsDefault** pragma is known as *implicit* or *transparent* mode, whereas defining a DirectToSOM class by inheriting from another DirectToSOM class (including **SOMObject**) is known as *explicit* mode.

By default, this option will be applied to all files compiled, with the exclusion of any compiler library header files. You can control to which files this option applies by using the **/Xs** option, discussed shortly.

/Gb

Syntax: **/Gb**[+ | -]

The **/Gb** option disables direct data access for SOM class data members. It is equivalent to specifying **#pragma SOMNoDataDirect** at the beginning of each compilation unit. The default is **/Gb-**.

If you use this option, every SOM class data member must be made into an attribute. See the discussion of the **SOMNoDataDirect** pragma earlier in this chapter for further details.

/Gz

Syntax: **/Gz**[+ | -]

The **/Gz** option is used to prevent early creation of SOM class objects and instead to create the class object on-demand at run time. By default, any static reference to a class (that is, one that uses the SOM class data structures **<className>ClassData** or **<className>CClassData**) will cause the compiler to create the corresponding class object as part of static initialization for the program. You can use this option to indicate that the class objects should not be created until needed at run time. Instead of creating the class object at static initialization time, the compiler will check before each static reference whether the class object has been created yet, and create it if necessary.

This option will add run-time overhead in the additional checking, but depending upon the number of classes you are using, this may save some run-time overhead in avoiding the creation of unnecessary class objects. You would typically use this option only if you have a relatively small number of references in your program to a large number of different classes.

This option applies only to statically referenced classes. By default, classes that are referenced dynamically will automatically be created when the containing DLL is loaded. (See Chapter 5, *Programming Considerations*, for further details regarding dynamically accessing SOM classes.)

/Xs

Syntax: **/Xs**<directory1; directory2; . . . >

The **/Xs** option is used to exclude all files in the given directory or directories from the implicit mode processing enabled by the **/Ga** option. This option allows you to globally specify implicit SOM mode for the compilation, but to exclude certain files.

Programming Considerations

This chapter covers the major differences between native C++ and DirectToSOM C++, and discusses various considerations for using DirectToSOM C++. In addition, some of the commonly encountered programming problems are highlighted and explained.

Differences between Native C++ and DirectToSOM C++

This section explores the major restrictions and semantic differences between native and DirectToSOM C++ classes. Some of the material was covered briefly in Chapter 2, but is expanded with more detail here.

Inheritance

SOM does not support an inheritance tree containing anything other than SOM classes. For DirectToSOM C++, this implies that a class hierarchy must contain all SOM or all native C++ classes; a mixed hierarchy is not

supported. SOM also does not permit multiple subobjects of the same type within an inheritance tree. The corresponding DirectToSOM rule is that a class may appear multiple times within a hierarchy only as a virtual base. In other words, only a single occurrence of each nonvirtual base class is allowed within a SOM hierarchy. Because this can affect how a program operates, the compiler will issue an error for each multiple occurrence of a nonvirtual base class in a SOM class hierarchy. **SOMObject** is a special case, as it is implicitly treated as a virtual base.

It is expected that a future release of the compiler will support a pragma that will allow multiple occurrences of nonvirtual base classes in the hierarchy. You will still only get a single subobject of that type in a given instance, but you will not be forced to make the base classes virtual, which does change the class semantics slightly. Essentially, the pragma would indicate that the SOM semantics will be used.

To illustrate these restrictions, the following shows various combinations of valid and invalid class hierarchies:

```

1  #include <som.hh>
2
3  class Som1 : public SOMObject {
4  };
5
6  class Som2 : public SOMObject {
7  };
8
9  // valid hierarchy: all SOM classes
10 class Som3: private Som1, protected Som2 {
11 };
12
13 // valid hierarchy: all SOM classes
14 class Som4: virtual public Som1, virtual private Som2 {
15 };
16
17 class nonSom1 {
18 };
19
20 // invalid hierarchy, mixing SOM and native
21 class mixed : public nonSom1, private Som1 {
22 };
23
24 // invalid hierarchy, Som2 non-virtual and appears twice
25 class Som5 : private Som3, public Som2 {
26 };
27
28 // invalid hierarchy, Som2 non-virtual in Som3, so appears twice
29 class Som6 : private Som3, virtual public Som2 {
30 };
31
32 // valid hierarchy, Som2 virtual in all bases
33 class Som7 : protected Som4, virtual public Som2 {
34 };

```

Inline Member Functions

Inline member functions are currently generated out-of-line by the Visual-Age C++ compiler so that RRBC won't be compromised. For example, if an inline member function accessed a private data member, any client in which that member function were inlined would require recompilation if a new private data member were added. The fact that member functions are generated out-of-line can greatly increase the size of binary objects over native C++ for applications that rely heavily on inlined functions.

Future releases of the compiler may permit inline functions where their inclusion in client code would not break RRBC, but it is unlikely that complete inlining will ever be supported.

sizeof

Because the size of a DirectToSOM C++ class is not known until run time, **sizeof** is a run-time constant expression for DirectToSOM instances and is not allowed in contexts that require compile-time evaluation. In the next example, the declaration of the integer `i` initialized to the size of type `A` is valid because the size of type `A` can be determined at run time in this context. But according to the C++ language rules, an array bound must be a positive integral constant expression, so `sizeof(A)` is not valid in the declaration of the array `arr`, because the compiler cannot evaluate the expression until run time.

```
#include <som.h>

class A : public SOMObject {
    int i;
};

int i = sizeof(A); // valid context for sizeof(A)

int arr[sizeof(A)]; // invalid context for sizeof(A)
```

Note that the value returned by **sizeof** is fixed within a given execution.

offsetof

offsetof depends upon the data member access, due to the reordering of data members described in Chapter 3. Public data starts at offset 0, while protected and private together start at offset 0 (with protected data first, followed by private data). For example, the result of running the program shown next, compiled without the macro `SOM` defined (that is, as native C++), is:

```
offset of priv: 0
offset of prot: 4
offset of pub: 8
```

and compiled with the macro `SOM` defined is:

```
offset of priv: 4
offset of prot: 0
offset of pub: 0
```

offsetof and Member Access (offsetof1.cpp):

```
1 #include <iostream.h>
2 #include <som.hh>
3
4 #ifdef SOM
5 #pragma SOMAsDefault(on)
6 #endif
7
8 class A {
9     private:
10         int priv;
11     protected:
12         int prot;
13     public:
14         int pub;
15         void displayOffset();
16 };
17
18 void A::displayOffset()
19 {
20     cout << "offset of priv: " << offsetof(A,priv) << endl
21         << "offset of prot: " << offsetof(A,prot) << endl
22         << "offset of pub: " << offsetof(A,pub) << endl;
23 }
24
25 int main(int arg, char *argv[])
26 {
27     A a;
28
29     a.displayOffset();
30 }
```

In addition, the offset is always relative to class that introduces it. Thus, `offsetof(base, base_element)` is always equal to `offsetof(derived, base_element)`. Therefore, you cannot use the offset of a data member within a class to get to the beginning of the instance data. For example, the result of running the next program compiled without the macro `SOM` defined, is:

```
offsetof(C,a) 0
offsetof(C,b) 4
offsetof(C,c) 8
```


and compiled with the macro SOM defined, is:

```
offsetof(C,a) 0
offsetof(C,b) 0
offsetof(C,c) 0
```

offsetof and Derived Class Member Access (offsetof2.cpp):

```
1  #include <iostream.h>
2  #include <stddef.h>
3
4  #ifdef SOM
5  #pragma SOMAsDefault(on)
6  #endif
7
8  struct A {
9      int a;
10 };
11
12 struct B {
13     int b;
14 };
15
16 struct C : public A, public B {
17     int c;
18 };
19
20 #ifdef SOM
21 #pragma SOMDefine(A)
22 #pragma SOMDefine(B)
23 #pragma SOMDefine(C)
24 #endif
25
26 int main(int argc, char *argv[])
27 {
28     C c;
29
30     cout << "offsetof(C,a) " << offsetof(C,a) << endl;
31         << "offsetof(C,b) " << offsetof(C,b) << endl;
32         << "offsetof(C,c) " << offsetof(C,c) << endl;
33 }
```

Address of a SOM Object

Because the size of a SOM object is not known until run time, SOM objects are implemented internally as pointers. In other words, given the declaration:

```
class A : public SOMObject {
};
A obj;
```

then `obj` is internally declared and manipulated as a pointer rather than an actual instance of class `A`. At run time, the storage is allocated for the object based on the class size, and the address of that storage is saved in the declared instance. For the most part, this implementation is transparent to the programmer. However, when the address of a `SOM` instance is taken, the compiler will return the actual pointer contained in the variable, rather than the address of the variable in which the pointer was stored. This is so that expressions such as:

```
A *ptr = &obj;
```

are handled correctly. In essence, a `SOM` instance behaves with respect to the client program as a reference to an object of that type. In other words, the declaration of `obj` in the preceding is essentially equivalent to:

```
A &obj;
```

When you take the address of a reference, you get the pointer contained in that reference, not the address of the reference variable itself. Note that these declarations are not equivalent in that there are some situations, such as template arguments, where `A` is valid but `A&` is not. Figure 5.1 illustrates this model.

This implementation becomes most noticeable when `SOM` members are embedded inside another class. Each embedded data member is implemented as a pointer, and storage for the each object is allocated at the end of the structure. Storage for the entire structure will be allocated at run time based on the total size of the embedded members. As previously described, when the address of such a member is taken, the address contained in the member of storage allocated at the end of the structure is returned rather than the address of the data member itself within the structure. Figure 5.2 illustrates this model for the following declarations:

```
class A : public SOMObject
{
    int i;
};

struct B {
    A a1;
    int j;
    A a2;
    int k;
} b;
```

This behavior results in one of the few areas where the `DirectToSOM` C++ support does not conform to the proposed ANSI C++ standard; the addresses of nonstatic `DirectToSOM` data members without intervening

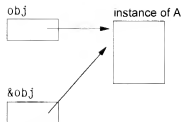


FIGURE 5.1 Implementation of DirectToSOM C++ objects.

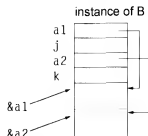


FIGURE 5.2 Class with embedded DirectToSOM C++ objects.

access specifiers are not strictly increasing within a class. For example, in the previous class definition, it is not the case that `&b.a1 < &b.j < &b.a2 < &b.k`; it is only true that `&b.j < &b.k`. When the program shown next is compiled without the macro `SOM` defined, the output on OS/2 is:

```
&b.a1: 0x40000
&b.j:  0x40004
&b.a2: 0x40008
&b.k:  0x4000c
```

and with the `SOM` macro defined:

```
&b.a1: 0x135790
&b.j:  0x50004
&b.a2: 0x1357b0
&b.k:  0x5000c
```

Address of a SOM Member (addrof.cpp):

```
1  #include <iostream.h>
2  #include <som.hh>
3
4  #ifdef SOM
5  class A : public SOMObject
6  {
7      int i;
8  };
9  #pragma SOMDefine(A)
10 #else
11 class A
12 {
13     int i;
14 };
15 #endif
16
17 struct B {
```

Continued

```

18     A a1;
19     int j;
20     A a2;
21     int k;
22 } b;
23
24 int main(int argc, char *argv[])
25 {
26     cout << "&b.a1: " << &b.a1 << endl
27         << "&b.j: " << &b.j << endl
28         << "&b.a2: " << &b.a2 << endl
29         << "&b.k: " << &b.k << endl;
30 }

```

Initializer List

You cannot use an initializer list to initialize DirectToSOM objects, because DirectToSOM classes always inherit from the **SOMObject** class. (An initializer list cannot be used to initialize an object of a class that has, among other things, a base class.) For example, both initializations in the next program are valid if **A** and **B** are not SOM classes, but the first initialization is invalid if both are SOM classes. The second is valid because the `B::B(int)` constructor can be called to construct each array element. (Actually, the second is currently flagged as an error in version 3.0 of the OS/2 product, but is valid in Windows. This will be valid in the next release of the OS/2 product.)

Initializer List (initlist.cpp):

```

1  #ifdef SOM
2  #pragma SOMAsDefault (on)
3  #endif
4
5  class A {
6  public:
7      int i;
8      int j;
9      int k;
10 };
11
12 // invalid if A is DirectToSOM
13 A a = {1, 2, 3};
14
15 class B {
16 public:
17     int i;
18     int j;
19     int k;
20     B(int input) { i=j=k=input; }
21 };
22

```

```

23 // valid if A is DirectToSOM
24 B arr[3] = {1, 2, 3};

```

Calling through a NULL Pointer

You cannot call a nonvirtual function through a NULL pointer to a DirectToSOM instance, because the method routing must be performed through the SOM API using a valid class instance. When the following program is compiled as non-DirectToSOM, it successfully invokes the `A::foo()` method, because the contents of the variable `a` are not required to determine the method to call—this can be determined statically by the compiler. However, when the program is compiled as DirectToSOM, a run-time exception will occur. This is because the object pointed to by `a` is used by the SOM API to perform method resolution. See Chapter 6, *Inside DirectToSOM C++*, for further details on the method resolution mechanism.

Calling through a Null Pointer (*callnull.cpp*):

```

1  #include <iostream.h>
2
3  #ifdef SOM
4  #pragma SOMAsDefault(on)
5  #endif
6
7  class A {
8  public:
9      void foo();
10 };
11
12 void A::foo()
13 {
14     cout << "In A::foo()" << endl;
15 }
16
17 int main(int argc, char *argv[])
18 {
19     A *a = NULL;
20
21     a->foo(); // invalid if A is DirectToSOM
22 }

```

Casting

You cannot cast a pointer-to-SOM object to arbitrary storage or an unrelated type (commonly referred to as class punning), because the SOM API depends upon the underlying object type. The following programming example will compile and run without error as native C++, correctly print-

ing out 5 for the value of *j*. However, an exception will occur if the classes are defined as `DirectToSOM`, even though the layout of the classes are identical.

Class Punning (punning.cpp):

```

1  #include <iostream.h>
2
3  #if SOM
4  #pragma SOMAsDefault(on)
5  #endif
6
7  struct A {
8      int i;
9      int j;
10 };
11
12 struct B {
13     int k;
14     int l;
15 };
16
17 #if SOM
18     #pragma SOMDefine(A)
19     #pragma SOMDefine(B)
20 #endif
21
22 int main(int argc, char *argv[])
23 {
24     A a;
25
26     ((struct B*)&a)->k = 5; // invalid for SOM
27     cout << "a.i: " << a.i << endl;
28 };

```

Linking

All static data and member functions must be defined by link time because they are used to construct the class tables. This is different from native C++, where such members do not need to be defined unless they are actually referenced in the program.

Programming Considerations

This section covers various programming paradigms and usage considerations that you should be aware of when developing applications with `DirectToSOM C++`.

Environment Parameter and Error Handling

The C++ exception-handling support does not map to the SOM model. When errors occur in invoking SOM methods, the SOM exception support must be used. The CORBA architecture requires that all methods accept a second argument that is of type **Environment** (the first argument being the address of the target object itself). The **Environment** structure is used to return exception information to the client following a method call. The declaration of this structure, from `<somcorba.h>`, is:

```
typedef struct Environment {
    exception_type          _major;
    struct {
        char *              _exception_name;
        void *              _params;
    } exception;
    void *                  _somedAnchor;
} Environment;
```

After a method invocation, the value of the `_major` field will be nonzero if an exception occurred. The rest of the structure is opaque, and not intended to be directly accessed by the programmer. Given that the `_major` field is nonzero after a method invocation, there are several functions that you can call to retrieve additional information. For example, `somExceptionID` returns the exception name, if any, as a string. (See the SOMObjects documentation for further details.)

If an exception does occur, it is imperative that the exception information contained in the **Environment** structure be deallocated, otherwise passing this **Environment** structure on subsequent method invocations can result in memory leaks and unexpected run-time errors.

The **Environment** structure is used heavily with DSOM and other such frameworks, but not very often for basic SOM support. For this reason, the DirectToSOM compiler implicitly passes an **Environment** structure to each method, but allows you to access the result if necessary.

The compiler implicitly passes the value of the variable `__SOMEnv` as the second argument to every method that is expecting it and defines `__SOMEnv` in the formal parameter list for that method. `__SOMEnv` is also implicitly declared by the compiler at file scope, so you don't need to declare it yourself. Following a method invocation, you can check the value of `__SOMEnv._major` to determine if an error occurred. For example:

```
obj.foo(); // __SOMEnv implicitly passed
if (__SOMEnv->_major != 0) {
    cout << "Exception: "
        << somExceptionID(__SOMEnv) << endl;
```

Continued

```

        somExceptionFree(__SOMEnv);
    }

```

The compiler will pass whichever **__SOMEnv** variable it finds at the current scope, according to the normal scope lookup rules. By default, this will be the file scope variable implicitly declared by the compiler. However, if you define a **__SOMEnv** variable in a local scope, the compiler will pass that one instead. Note that you cannot define your own version of **__SOMEnv** at function scope inside a SOM method, because it is already declared at that scope as a formal parameter. But, as with any other formal parameter, you can assign a new **Environment** address to that incoming parameter.

When you call a SOM method from within a SOM method, the **__SOMEnv** parameter received by the enclosing method is passed on the nested method invocation, rather than the global **__SOMEnv**. As an example, in the following program, the copy of **__SOMEnv** passed to both `foo` at line 22, and to the subsequent call to `bar` at line 9, will be that defined in the main function at line 19.

Using __SOMEnv (env.cpp):

```

1  #include <iostream.h>
2  #include <somd.hh>
3
4  // __SOMEnv implicitly declared by compiler
5  static Environment ev =
6      (SOM_InitEnvironment(__SOMEnv = &ev), ev);
7
8  struct A : public SOMObject {
9      void foo() { bar(); }
10     void bar() { }
11     #pragma SOMDefine(*)
12 };
13
14 int main(int argc, char *argv[])
15 {
16     A obj;
17
18     // hides global __SOMEnv
19     Environment ev, *__SOMEnv = &ev;
20     SOM_InitEnvironment (__SOMEnv);
21
22     obj.foo(); // local __SOMEnv implicitly passed
23     if (__SOMEnv->major != 0) {
24         cout << "Exception: "
25              << somExceptionId(__SOMEnv) << endl;
26         somExceptionFree(__SOMEnv);
27     } else {
28         cout << "Success" << endl;
29     }
30 }

```


Callstyle: IDL and OIDL

As discussed in Chapter 4 in the `SOMCallStyle` section, earlier versions of SOM did not support the **Environment** parameter, so methods of classes defined using these earlier versions do not accept an **Environment** parameter. Methods that expect the **Environment** parameter are classified as callstyle **IDL**, whereas those that don't are callstyle **OIDL**.

For methods that have callstyle **OIDL**, exception information is communicated through a global **Environment** structure, which can be accessed through the `somGetGlobalEnvironment` function. This **Environment** structure is shared by all objects running in the same thread.

The classes **SOMObject** and **SOMClass** have callstyle **OIDL**, as they were defined prior to the availability of **Environment** support. All methods of these classes, in particular, the 10 special **SOMObject** methods that map to C++ methods, do not accept an **Environment** parameter. Therefore, when these methods are overridden in a C++ class, they do not expect an **Environment** parameter. But any introduced constructors that do not map to one of these methods will have the callstyle of the defining class.

For example, in the following class definition, `A()` will have callstyle **OIDL**, because it maps to `somDefaultUnit`, which is introduced in **SOMObject**, which is callstyle **OIDL**. However `A(int)` will have callstyle **IDL**, because it does not override or map to any **SOMObject** methods and therefore has the callstyle of the defining class.

```
struct A : public SOMObject {
    A() {}
    A(int) {}
};
```

To check for an error after creating an object with `A()`, you would check the global **Environment** structure; and with `A(int)`, you would check `__SOMEnv`. In general, you don't need to check for errors after such methods invocations, but if you find that you need to, for simplicity you could set `__SOMEnv` as the global **Environment** structure and then check either one for all method invocations.

Initializing `__SOMEnv`

Earlier versions of the compiler, including the version 3.0 products for both OS.2 and Windows, that I was using declared `__SOMEnv`, but did not initialize it. Future releases of the compiler will declare and initialize the variable. You must ensure that this variable is initialized before using it, otherwise a method that attempts to assign exception information will fail. Some SOM methods check that the **Environment** structure is valid before proceeding. In order to ensure that `__SOMEnv` is initialized, there are a couple of options

available. One possibility is to include the following line in the beginning of the file:

```
static Environment ev =
    (SOM_InitEnvironment(&__SOMEnv, ev));
```

This will declare a variable, `ev`, of type **Environment**, assign it to `__SOMEnv`, and initialize it by calling **SOM_InitEnvironment**. Because `ev` is declared as static, this will take place as part of static program initialization. If you have any SOM methods called as part of static program initialization, you will need to make sure that the initialization of `__SOMEnv` takes place first.

Another option is to initialize `__SOMEnv` using the **SOM_CreateLocalEnvironment** function as follows:

```
__SOMEnv = SOM_CreateLocalEnvironment ();
```

This will dynamically allocate and initialize an **Environment** structure and assign the result to `__SOMEnv`. Since this is dynamically allocated storage, you must remember to deallocate it using the **SOM_DestroyLocalEnvironment** function:

```
SOM_DestroyLocalEnvironment (__SOMEnv);
```

Templates

DirectToSOM C++ template classes are defined in much the same way as normal template classes, with the exception that they inherit from a SOM class. DirectToSOM C++ supports template class definitions by mapping each unique instantiation to a SOM class definition. Templates are instantiated in the same way as for native C++, either by declaring a variable of that type, creating a type definition for that type, or through the **#define** pragma. The simplest way to handle template instantiation with DirectToSOM (and with native C++) is to use the automatic template generation support provided by the compiler.

With automatic template generation, the compiler builds a *template-include* file that contains information about instantiated template classes. This file is not compiled until just before the compiler invokes the linker. Each template instantiation is thus compiled only once per a given program, rather than into each object file where it is instantiated. This is particularly important for SOM classes, because you do not want the SOM class data structures to be generated in multiple object files. Automatic template generation also allows the template class implementation to be updated without recompiling client code.

To use automatic template generation, put the template class definition in an **.hh** file, and the template member definitions in a **.e** file of the same name. Then compile any source files with the **/c** option to generate objects. For each file compiled, the compiler will generate a corresponding **.cpp** template-include file in the **TEMPINC** directory (if the **TEMPINC** environment variable is not set, the compiler will generate a **TEMPINC** directory in the current directory). When the object files are subsequently compiled, the compiler will compile the necessary class definitions that were generated into the **TEMPINC** directory. See the VisualAge C++ documentation for further details.

The following programming example shows how to define, build, and use a **DirectToSOM** template class. The class is defined as a normal template class, with the exception that it is derived from **SOMObject**. The class definition is fairly straightforward, too. The class **Stack<int>** is instantiated at line 6 in the client program. Note that when compiling the **.obj** files, the **/Tdp** option must be specified. If the file type is not specified as C++, the compiler will not generate the template include files and will not compile and link the template class definitions. This option is not necessary when compiling the **.cpp** files because the language is implied by the file extension. I have included only one makefile, because the same one is valid for both OS/2 and Windows.

Definition of DirectToSOM Template Class (template\stack.hh):

```

1  #include <som.hh>
2
3  template <class T> class Stack : public SOMObject {
4      T *stack;
5      int top;
6      int arraySize;
7  public:
8      Stack(int) ;
9      ~Stack() ;
10     void push(T) ;
11     T pop() ;
12     int size() ;
13 } ;
```

Implementation of DirectToSOM Template Class (template\stack.c):

```

1  #include <assert.h>
2
3  template <class T> Stack<T>::Stack(int s)
4  {
5      top = 0;
6      stack = new T[arraySize = s] ;
7      assert(stack) ;
```

Continued

```

8  }
9
10 template <class T> Stack<T>::~Stack()
11 {
12     if (stack)
13         delete stack;
14 }
15
16 template <class T> void Stack<T>::push(T elem)
17 {
18     if (top < arraySize)
19         stack[top++] = elem;
20 }
21
22 template <class T> T Stack<T>::pop()
23 {
24     if (top)
25         return stack [--top];
26     return 0;
27 }
28
29 template <class T> int Stack<T>::size()
30 {
31     return top;
32 }

```

Client of DirectToSOM Template Class (template\main.cpp):

```

1  #include <iostream.h>
2  #include "stack.hh"
3
4  int main(int argc, char *argv[])
5  {
6      Stack<int> intStack(100) ;
7
8      intStack.push(4) ;
9      intStack.push(3) ;
10     intStack.push(2) ;
11     intStack.push(1) ;
12     cout << "size: " << intStack.size() << endl;
13     cout << intStack.pop() << endl;
14     cout << intStack.pop() << endl;
15     cout << intStack.pop() << endl;
16     cout << intStack.pop() << endl;
17     cout << "size: " << intStack.size() << endl;
18 }

```

Makefile (template\makefile):

```

1  all: main.exe
2
3  # need -tdp when compile .obj

```

```

4  main.exe: main.obj
5      gcc -tdp main.obj
6
7  main.obj: stack.hh main.cpp
8      gcc -c main.cpp

```

Using the **DirectToSOM** Pragas with Template Classes

Because each template instantiation maps to an individual SOM class definition, if you want to assign a SOM class name to a class, you must do so for each instantiation of that class. This is achieved by specifying the template class name with the **SOMClassName** pragma. For example,

```

#pragma SOMClassName (Stack<int>, "IntegerStack")

#pragma SOMClassName (Stack<char *>, "StringStack")

```

With template classes, you cannot use the `SOMClassName(*, "SOMName")` notation, because every instantiation would be assigned the same name. This would work for the first instantiation, but would result in an error for the second because you cannot define more than one class with the same name.

The **SOMDefine** pragma gets a bit tricky when using templates. It is very easy to end up with multiple definitions of the class across various objects. If you do have a class that has no out-of-line member functions, I recommend that you define a dummy out-of-line member function and use the automatic template generation support rather than use **SOMDefine** to force the generation of the class data structures.

The **SOMReleaseOrder** pragma is specified for template classes in the same way as for normal classes. You can use the template arguments to specify member function signatures types. For example, the release order for the preceding class is:

```

#pragma SOMReleaseOrder (Stack<int>, push(T), pop(), size() )

```

The template argument can be used only for pragmas that occur within the scope of the class template definition. In order to use the **/Fr** option to generate the release order list for a template class, you must currently use an instantiated class, and then backtrack to remove the instantiation-specific information. For example, I generated the preceding release order statement using `/Fr Stack<int>` and changed the occurrences of `int` to `T`.

Memory Management

The **DirectToSOM** compiler includes additional versions of the **new** and **delete** operators that are called for SOM objects. These operators have the following signatures:

```

void operator ::new(SOMClass *, size_t) ;

void operator ::new(SOMClass *, size_t, void *location) ;

void operator ::new[](SOMClass *, size_t) ;

void operator ::new[](SOMClass *, size_t, void *location) ;

void operator ::delete(SOMObject *, size_t) ;

void operator ::delete[](SOMObject *, size_t) ;

```

When storage is dynamically allocated or deleted for a DirectToSOM C++ object using **new** or **delete**, these SOM-enabled operators are called rather than the standard **new** and **delete** operators. In addition, debug versions of these operators are supplied for using the debug memory management support, available when you compile a program with **/Tm+** in OS/2 or Windows. You must include **<somnew.h>** into your program to use the versions with the location placement arguments (these versions allocate the object in a supplied storage location).

The SOM versions of operator **new** and **delete** call the **SOMMalloc** and **SOMFree** memory management routines supplied by the SOM run time, so that all DirectToSOM objects by default are allocated through the SOM memory management routines. **SOMMalloc** and **SOMFree** provide common memory management routines that can be used by any compiler for allocating and deallocating SOM objects. Having a common memory manager allows objects allocated through DirectToSOM C++ to be deallocated by a program written using the C++ language bindings and compiled with a completely different compiler. This would not be possible if each compiler product used its own memory management routines.

Objects with automatic storage duration are allocated at run time on the stack using the **alloca** library function, which does not call **SOMMalloc** (see Chapter 6, for details). This is not a problem, because such objects cannot be dynamically deallocated anyway. Objects with static storage duration are allocated and deallocated with the SOM versions of **new** and **delete**, as are dynamically allocated objects.

Overloading new and delete

You can overload the compiler-supplied versions of the SOM **new** and **delete** operators by providing custom versions with the preceding signatures. In addition, placement arguments may be added after the **size_t** parameter following the standard C++ rules. The **SOMClass *** argument is always passed implicitly as a pointer to the class object corresponding to the instance being allocated. There is no provision for the programmer to supply a different value for the class object parameter.

For example, the following program shows how to overload the **new** operator for SOM classes. At lines 5 through 10, the SOM operator **new** is overloaded with the identical signature as the compiler-supplied version, while at lines 12 through 22, it is overloaded with a placement argument that is used to initialize the allocated storage. The storage allocation for class A at line 34 will invoke the version at line 5, and the allocation for class B at 35 will invoke the version at line 12. The output of this program is:

```
allocating 4 for a za
```

```
allocating 4 with init # for a zb
```

Overloading Global new (newglob.cpp):

```

1  #include <iostream.h>
2  #include <assert.h>
3  #include <som.hh>
4
5  void* operator new (SOMClass *classObject, size_t size)
6  {
7      cout << "allocating " << size << " for a "
8          << classObject->somGetName() << endl;
9      return SOMMalloc(size);
10 }
11
12 void* operator new(SOMClass *classObject, size_t size,
13                   char init)
14 {
15     cout << "allocating " << size << " with init "
16         << init << " for a "
17         << classObject->somGetName() << endl;
18     void *tmp = SOMMalloc(size);
19     assert(tmp);
20     memset(tmp, size, init);
21     return tmp;
22 }
23
24 class A : public SOMObject {
25     #pragma SOMDefine(*)
26 };
27
28 class B : public SOMObject {
29     #pragma SOMDefine(*)
30 };
31
32 int main(int argc, char *argv[])
33 {
34     A *a = new A;
35     B *b = new('#') B;
36
37     delete a;

```

Continued

```

38     delete b;
39 }

```

A note regarding syntax: In standard C++, the first argument to **new** must be a **size_t**. Because the first argument to the SOM-enabled operator **new** is **SOMClass ***, it is distinct from a standard operator **new** that has a placement argument of **SOMClass ***, such as:

```
void operator ::new(size_t, SOMClass *) ;
```

The same is true of the SOM-enabled operator **delete**. This is probably not something you will ever have to worry about, but it's worth mentioning.

Providing Class-Specific new and delete

In addition to overloading the global SOM **new** and **delete** operators, you can also provide class-specific **new** and **delete** operators, as with standard C++. The signatures are the same as for the global versions, but the initial **SOMClass *** or **SOMObject *** parameters are optional, because the compiler does not need them in the signature to distinguish between the standard and SOM versions. You can, however, only supply one version of the class-specific operator, with or without the optional parameter. Note that it is possible to call an operator explicitly:

```
A::operator delete(object, size)
```

so when overriding operator **delete**, you may want to invoke the **somIsObj** function to verify that the pointer supplied is actually a SOM object before deleting the storage.

To illustrate, I have modified the previous example to supply class-specific versions of operator **new** for the classes **A** and **B**, in addition to providing a global override. The storage allocation at line 44 for class **A** will invoke the class-specific version of **new** for class **A** at line 17, as the allocation for class **B** at line 45 will invoke the class-specific **new** for class **B** at line 28. The allocation for class **C** at line 46 will invoke the global operator **new** overload at line 5. The output of this program is:

```

allocating 4 for an A
allocating 4 with init # for a B
allocating 4 for a c with global new

```

Overloading Class-Specific new (newclass.cpp):

```

1  #include <iostream.h>
2  #include <assert.h>
3  #include <som.hh>
4

```



```

5 void* operator new(SOMClass *classObject, size_t size)
6 {
7     cout << "allocating " << size << " for a "
8         << classObject->somGetName()
9         << " with global new" << endl;
10    return SOMMalloc(size);
11 }
12
13 struct A : public SOMObject {
14     void* operator new(SOMClass *classObject, size_t size) ;
15 };
16
17 void* A::operator new(SOMClass *classObject, size_t size)
18 {
19     cout << "allocating " << size
20         << " for an A" << endl;
21     return SOMMalloc(size) ;
22 }
23
24 struct B : public SOMObject {
25     void* B::operator new(size_t size, char init) ;
26 };
27
28 void* B::operator new(size_t size, char init)
29 {
30     cout << "allocating " << size << " with init "
31         << init << " for a B" << endl;
32     void *tmp = SOMMalloc(size) ;
33     assert(tmp);
34     memset(tmp, size, init) ;
35     return tmp;
36 }
37
38 class C : public SOMObject {
39     #pragma SOMDefine(*)
40 };
41
42 int main(int argc, char *argv[])
43 {
44     A *a = new A;
45     B *b = new('#') B;
46     C *c = new C;
47
48     delete a;
49     delete b;
50     delete c;
51 }

```

SOMMalloc and SOMFree

If you do overload **new** or **delete** operators to call something other than the SOM memory routines, you must ensure that this is done consistently; **new**, **new []**, **delete**, and **delete []** must all be overridden to use the same

memory manager. (See the header file `<somnew.h>` for signatures of all the SOM memory management routines.) Storage allocated with **SOMMalloc** must be freed by **SOMFree**, and **SOMFree** can only deallocate storage allocated by **SOMMalloc**. If, in the previous examples, I had called **malloc** instead of **SOMMalloc**, the **delete** operations would all fail because they would still call **SOMFree**, which would attempt to deallocate storage allocated by the C run-time library routines. This will not necessarily result in an exception; depending upon the memory manager used, the deallocation may just quietly fail to deallocate the storage, resulting in memory leaks.

Even without overloading **new** and **delete**, you must be aware of memory allocator mismatches in situations such as using the placement syntax to allocate an object in a supplied piece of storage. You cannot, for example, use **malloc** to allocate storage for a **SOMObject**, and then use **delete** to deallocate that storage, because a **delete** operation against a **SOM** object will invoke **SOMFree** to deallocate the storage. Be aware also that the size of a **SOM** object is the size of the instance data plus the extra pointer at the beginning (see Chapter 6, *Inside DirectToSOM C++*, for details). If you are using the location placement argument, you should ensure that storage is allocated using **sizeof** the **SOM** class type, rather than what you would expect the instance data size would be.

The next programming example illustrates these programming errors. At line 12, 4 bytes of storage are allocated using **malloc** and used to place an instance of class **A** at line 13. This is incorrect because an instance of class **A** will be 8 bytes (assuming 4 bytes for a pointer and 4 bytes for an integer). The correct approach would be to use `sizeof(A)` instead of 4 with **malloc**. A second error is that **delete** is used to deallocate the storage, which will call **SOMFree**, because **a** is a **SOM** object, rather than the standard C++ deallocation routine. A correct approach in this example would be to allocate the storage using **SOMMalloc**. Note that deleting the storage using **free** would be correct in this simple example, but not in general, because the class destructor would not be called.

Invalid Memory Management Operations (newbad.cpp):

```

1  #include <stdlib.h>
2  #include <som.hh>
3  #include <somnew.h>
4
5  struct A : public SOMObject {
6      int i;
7      #pragma SOMDefine(*)
8  };
9
10 int main(int argc, char *argv[])
11 {
```

```

12 void *ptr = malloc(4) ;
13 A *a = new(ptr) A; // error, not large enough
14 delete a;         // error, uses SOMFree
15 }

```

In addition, if you do allocate SOM objects with something other than **SOMMalloc**, other programs will not be able to deallocate these objects unless they also have access to, and use, the same memory management routines. This may not be a problem if the memory you allocate will never be deallocated outside of your application, but it is something to consider.

In certain situations, such as when using DSOM, SOM will use **SOMMalloc** to allocate and return memory for items such as strings. This storage cannot be deallocated with the **delete** operator, which will call the C++ memory management routines, as it must be freed instead with **SOMFree**. Likewise, situations will arise where SOM will deallocate storage for such items using **SOMFree**. The storage must have been allocated with **SOMMalloc** for this operation to be successful. For such applications, rather than keep track of which allocator owns the memory, it is much simpler to either manage all memory through **SOMMalloc/SOMFree** or through the C++ memory management routines. In Chapter 8, we will discuss this problem in more detail and provide examples of how to deal with it.

SOM DLLs

There are two ways to load a SOM DLL, either statically or dynamically. The DLL will be loaded statically if the client contains any static references to the SOM class. By default, objects are manipulated statically through DirectToSOM, as they are through the C and C++ language bindings. All the examples so far have access to the SOM class symbols statically. Later in this section, I will show how to load a class dynamically with DirectToSOM.

It is important that a SOM DLL be constructed so that it can be loaded statically or dynamically, as it is often difficult to predict how a class library will be used. For example, in order to use a class library from DSOM or Smalltalk, it must be possible to dynamically load that library.

Because each platform has different mechanisms for creating and loading DLLs, I will concentrate on OS/2 and Windows in the following discussion. The basic concepts are the same for all platforms, but the commands and mechanisms involved may differ. In order to access a class statically, the SOM class definition must export the three class symbols mentioned in earlier chapters. These are **<class>ClassData**, **<class>CClassData**, and **<class>NewClass**. When creating a DLL containing a SOM class that will be accessed statically, these symbols must be exported from the DLL. You do not need to export any other symbols, such as the method names themselves, as the SOM class symbols provide the sole means through which the SOM class is manipulated.

Exporting Static DLL Symbols Using `_Export`

There are a variety of means by which you can export these symbols from a DLL. In OS/2 or Windows, you can specify the `_Export` keyword with the SOM class definition, which will cause the compiler to automatically export the three symbols from any DLL. The next example creates a DLL for the class `Hello`. The symbols are automatically exported from the DLL by the presence of the `_Export` keyword in the class definition file at line 6.

In addition to specifying which symbols are to be exported from a DLL, with Windows you must also explicitly specify symbols that are to be imported from a DLL. There are several ways to do this: the `_Import` keyword, the `_declspec(dllexport)` specifier, or the `-qautoimport` compiler option. The use of the `_Import` keyword is shown at line 4 in the class definition file. If the macro `WINCLIENT` is defined, then the presence of the `_Import` keyword will cause the compiler to generate special code to indicate that the class symbols are to be imported from another DLL. I will illustrate the use of `_declspec(dllexport)` in the next example, where `qautoimport` is used to indicate that all external references are to be imported from a DLL, rather than specifying anything extra in the source code itself. This option may add unnecessary overhead to the generated code—see the compiler documentation for details.

In the OS/2 makefile, the DLL is created at line 4. In order to create a DLL for OS/2, a minimal module definition file supplying the DLL name is required, as shown in file `hello.def`. The `/Ge-` option indicates that a DLL is being created. Once the DLL is created, an import library file is created at line 5. This produces the file `hello.lib`, which is linked to the client program at line 8. The `hello.lib` file will contain an import entry for each symbol exported from `hello.dll`. (You can use the `ILIB` utility in OS/2 to view the contents of this file. Type `ILIB hello.lib`, hit Enter at the options prompt, and then type an output file name at the listing file prompt. This file will contain all the symbols exported from the DLL.)

When creating a DLL for Windows, a module definition file is not necessary if you have used `_Export` to define the exports. When the `/Ge-` option is specified in such situations, the compiler will automatically create the corresponding `.lib` file for the DLL, so an extra command is not necessary to generate it. (To view the contents of the generated `.lib` file in Windows, use the `ILIB` utility, specifying an option of `/list`. For example: `ILIB /list hello.lib`.)

When the client program is compiled at line 8 in the Windows makefile, the macro `WINCLIENT` is defined so that the class symbols will be properly imported from the target DLL.

Definition of Class `Hello` (`somdllnos2\export\hello.hh`):

```
1 #include <som.hh>
2
```

```

3  #ifdef WINCLIENT
4  class _Import Hello : public SOMObject {
5  #else
6  class _Export Hello : public SOMObject {
7  #endif
8      public:
9          void sayHello() ;
10 };
11

```

Implementation of Class Hello (somdllvos2\export\hello.cpp):

```

1  #include <iostream.h>
2  #include "hello.hh"
3
4  void Hello::sayHello()
5  {
6      cout << "Hello world" << endl;
7  }

```

Client of Class Hello (somdllvos2\export\tsthellob.cpp):

```

1  #include "hello.hh"
2
3  int main(int argc, char *argv[])
4  {
5      Hello obj;
6
7      obj.sayHello() ;
8  }

```

OS/2 Makefile (somdllvos2\export\makefile):

```

1  all: hello.dll tsthello.exe
2
3  hello.dll: hello.hh hello.cpp
4             gcc /Ge- hello.cpp hello.def
5             implib hello.lib hello.dll
6
7  tsthello.exe: tsthello.cpp hello.lib
8             gcc tsthello.cpp hello.lib

```

OS/2 Module Definition File (somdllvos2\export\hello.def):

```

1  LIBRARY hello INITINSTANCE
2  DESCRIPTION 'Hello Class Library'
3  PROTMODE
4  DATA MULTIPLE NONSHARED LOADONCALL

```

Windows Makefile (somdll\win\export\makefile):

```

1  all: hello.dll tsthello.exe
2

```

Continued

```

3 # icc will generate .libile
4 hello.dll: hello.hh hello.cpp
5         icc /Ge- hello.cpp
6
7 tsthello.exe: tsthello.cpp hello.lib
8         icc -DWINCLIENT tsthello.cpp hello.lib

```

Exporting Static DLL Symbols Using Manually Generated .def File

Another mechanism by which the SOM class symbols can be exported from a DLL is by creating a **.def** file that contains the appropriate symbols and linking it to the DLL. Using the previous example, I updated the class definition file and the **.def** file. Instead of using the **_Export** keyword to export the symbols, I instead supplied them explicitly in the **.def** file. I added a **SOMClassName** pragma just to make the symbols names more readable. In addition, I used the **_declspec(dllexport)** modifier for defining symbols imported from a Windows DLL. I find this method a little more readable than **_Import** when guarding with a macro, but they are equivalent.

Because a **.def** file is being supplied, in Windows we must now explicitly generate the **.lib** file. This is done through the **ILIB** command with option **/geni**, shown in the Windows makefile at line 5. This generates a **.lib** and an **.exp** file corresponding to the supplied **.def** file. It is expected that a future release of the OS/2 product will also support the **ILIB** command, with the same options currently supported for Windows, rather than the **IMPLIB** command.

In OS/2, the symbols are specified in the **.def** file directly, whereas for Windows they must be prepended with an **'_'**.

Definition of Class Hello (somedll\mandef\os2\hello.hh):

```

1 #include <som.hh>
2
3 #ifdef WINCLIENT
4 _declspec(dllexport)
5 #endif
6 class Hello : public SOMObject {
7     #pragma SOMClassName (*, "Hello")
8     public:
9         void sayHello();
10 };

```

OS/2 Makefile (somedll\mandef\os2\makefile):

```

1 all: hello.dll tsthello.exe
2

```

```

3 hello.dll: hello.hh hello.cpp
4         icc /Ge- hello.cpp hello.def
5         implib hello.lib hello.dll
6
7 tsthello.exe: tsthello.cpp hello.lib
8         icc tsthello.cpp hello.lib

```

OS/2 Module Definition File (somedll\mndef\os2\hello.def):

```

1 LIBRARY hello INITINSTANCE
2 DESCRIPTION 'Hello Class Library'
3 PROTMODE
4 DATA MULTIPLE NONSHARED LOADONCALL
5 EXPORTS
6     HelloClassData
7     HelloCClassData
8     HelloNewClass

```

Windows Makefile (somedll\mndef\win\makefile):

```

1 all: hello.dll tsthello.exe
2
3 hello.dll: hello.hh hello.cpp
4         icc /Ge- hello.cpp hello.def
5         ilib /geni hello.def
6
7 tsthello.exe: tsthello.cpp hello.lib
8         icc -DWINCLIENT tsthello.cpp hello.lib

```

Windows Module Definition File (somedll\mndef\win\hello.def):

```

1 LIBRARY hello
2 EXPORTS
3     _HelloCClassData
4     _HelloClassData
5     _HelloNewClass@8

```

Exporting Static DLL Symbols Using Automatically Generated .def File

A third option for exporting SOM DLL symbols, which I find most convenient to use, is to have the SOM compiler automatically generate the module definition file for you. In this example, I modified the makefile file to automatically generate the **.def** file. In order to do this, you must generate the IDL for the file. IDL generation is covered in detail in Chapter 7, but for now, suffice it to say that you can generate an IDL file from a DirectToSOM class definition file by compiling the **.hh** file directly, as shown in the OS/2 makefile at line 11. This will generate a corresponding **.idl** file. The **.def** file is then generated using the SOM **def** emitter, which is executed through the

sc command, specifying the **-sdef** option, as shown at line 12 in the OS/2 makefile. This will generate a corresponding **.def** file containing the appropriate symbol exports, as shown.

The Windows makefile essentially follows the same process. Note that I have qualified the path name for the **sc** command because the name may conflict with a Windows NT command name on many systems.

Definition of Class Hello (somdll\autodef\os2\hello.hh):

```

1  #include <som.hh>
2
3  #ifdef WINCLIENT
4  _declspec (dllimport)
5  #endif
6  class Hello : public SOMObject {
7      #pragma SOMClassName(*, "Hello")
8      public:
9          void sayHello() ;
10 };

```

OS/2 Makefile (somdll\autodef\os2\makefile):

```

1  all: hello.dll tsthello.exe
2
3  hello.dll: hello.hh hello.cpp hello.def
4      icc /Ge- hello.cpp hello.def
5      implib hello.lib hello.dll
6
7  tsthello.exe: tsthello.cpp hello.lib
8      icc tsthello.cpp hello.lib
9
10 hello.def: hello.hh
11      icc hello.hh
12      sc -sdef hello.idl

```

OS/2 Generated Module Definition File (somdll\autodef\os2\hello.def):

```

1  ; This file was generated by the SOM Compiler.
2  ; FileName: hello.def.
3  ; Generated using:
4  ;     SOM Precompiler somipc: 2.29.1.16
5  ;     SOM Emitter emitdef: 2.47
6  LIBRARY hello INITINSTANCE
7  DESCRIPTION 'Hello Class Library'
8  PROTMODE
9  DATA MULTIPLE NONSHARED LOADONCALL
10 EXPORTS
11     HelloClassData
12     HelloCClassData
13     HelloNewClass

```


Windows Makefile (somdll\autodef\win\makefile):

```

1  all: hello.dll tsthello.exe
2
3  hello.dll: hello.hh hello.cpp hello.def
4      gcc /Ge- hello.cpp hello.def
5      ilib /geni hello.def
6
7  tsthello.exe: tsthello.cpp hello.lib
8      gcc -DWINCLIENT tsthello.cpp hello.lib
9
10 hello.def: hello.hh
11      gcc hello.hh
12      $(SOMBASE)\bin\sc -sdef hello.idl

```

Windows Generated Module Definition File (somdll\autodef\win\hello.def):

```

1 ; This file was generated by the SOM Compiler.
2 ; FileName: hello.def.
3 ; Generated using:
4 ;     SOM Precompiler somipc: Development
5 ;     SOM Emitter emitdef.dll: Development
6 LIBRARY hello
7 EXPORTS
8     _HelloCClassData
9     _HelloClassData
10    _HelloNewClass@8

```

Enabling a DLL for Dynamic Loading

Before a SOM object can be created, the corresponding SOM class object must be created. This is achieved by calling the **<className>NewClass** routine. Clients that manipulate a class statically will implicitly call this routine to create the class object, so you should never need to worry about it. (Chapter 6 will describe in more detail how this takes place for Direct-To-SOM.)

For clients that load a SOM DLL dynamically, the class object must be created as part of loading the DLL, so that once the DLL is loaded, the class object is ready to be used. In SOMObjects 3.0, this is achieved through supplying a customized version of the DLL init/term functions. The simplest way to do this is through the SOM compiler **imod** emitter, using the **sc** command with option **-simod**, which generates an init/term function from the IDL. This init/term function is then linked into the DLL to correctly perform initialization when the DLL is dynamically loaded.

When SOM loads a class dynamically, it looks in one of its databases, called the *interface repository*, for the name of the DLL containing that class. If the DLL name is found, it then loads the named DLL, which will cause the class objects to be initialized. In order to generate the DLL name into the

interface repository, you must do two things. The first is to supply a special modifier, **dllname**, that will be generated into the IDL file, and the second is to register the class in the interface repository using the **sc** command with the options **-sir -u**.

The following example extends the previous class definition and makefile to create a SOM DLL that can be both statically and dynamically loaded. The **SOMIDLPass** pragma at lines 5 and 6 in the class definition file causes the IDL modifier:

```
dllname = "hello.dll"
```

to be generated into the IDL file. At line 13 in the makefile, the generated IDL definition is registered in the interface repository. In Chapter 7, I will explain IDL generation and interface repository in more detail. But for now, this simply generates information so that the SOM run time can look up the class Hello and find the name of the DLL that contains it, hello.dll. At line 15 in the makefile, the command:

```
sc -simod hello.idl
```

generates the DLL init/term function into the file hello.i.c. This file is rather large, and not particularly interesting, so I didn't include it here. The hello.i.c file is then compiled with the DLL at line 3 in the makefile. Because the compiler already supplies an init/term function of the same name (**_DLL_InitTerm** for OS/2 and Windows) the **/B"/NOE** option is used to when creating the DLL to indicate that the user-supplied version should be used rather than the compiler-supplied one. Without this option, a duplicate-definition link error would occur:

Definition of Class Hello (somedll\os2\dyns30\hello.hh):

```
1 #include <som.hh>
2
3 class Hello : public SOMObject {
4     #pragma SOMClassName(*, "Hello")
5     #pragma SOMIDLPass(*, "Implementation-End", \
6         "dllname = \"hello.dll\";");
7     public:
8         void sayHello() ;
9 };
```

Makefile (somedll\os2\dyns30\makefile):

```
1 all: hello.i.c hello.dll tsthello.exe
2
3 hello.dll: hello.hh hello.cpp hello.i.c
```

```

4      gcc /Ge- /B"/NOE" \
5          hello.cpp hello.i.c hello.def
6      implib hello.lib hello.dll
7
8      tsthello.exe: tsthello.cpp hello.lib
9      gcc tsthello.cpp hello.lib
10
11     hello.i.c: hello.hh
12         gcc hello.hh
13         sc -sir -u hello.idl
14         sc -sdef hello.idl
15         sc -simod hello.idl

```

Enabling a DLL for Dynamic Loading in SOM 2.x

With SOM 2.x, dynamic loading is achieved by supplying a **SOMInitModule** routine for the DLL that invokes the appropriate **NewClass** routines. When the SOM run-time loads a DLL dynamically, it calls this routine to ensure that the class objects were initialized properly. This approach has some problems, however, one of them being that the **SOMInitModule** function needs to be exported from the DLL, but should not appear in the corresponding **.lib** file. The reason is that multiple **.lib** files, each containing a **SOMInitModule** entry, cannot be combined into a single **.lib** file. Well, actually, they can, but an error will be generated if you actually try to use that **.lib** file. The SOM 3.0 approach circumvents this and other problems with DLL activation, and provides a system-independent mechanism for loading and initializing SOM DLLs.

The SOMObjects 3.0 beta was used to test most of the examples in the book where dynamic loading was required, specifically, those in Chapters 8 and 10. Some examples, specifically the interlanguage sharing examples in the Chapter 9, were tested using SOM 2.1, and therefore use this earlier approach for dynamically loading SOM DLLs.

The following programming example shows how to create a dynamically and statically loadable SOM DLL using the 2.x method for OS/2 and Windows. The main difference from the previous example is that rather than have the **imod** emitter generate the **hello.i.c** file, the function **SOMInitModule** is supplied. **SOMInitModule** will be called by the SOM run-time when the DLL is loaded, which will in turn call the **HelloNewClass** function to create the **Hello** class object.

The **SOMInitModule** function must be exported from the DLL, which is why the **EXPORT** macro is used. In addition, the linkage of the function must be specified correctly using the **SOMEXTERN** and **SOM-LINK** macros in order for the SOM run time to be able to invoke it. The **.lib** file produced in this example will contain an export for the **SOMInitModule** function, so if you will be combining the **.lib** files, you should not define the **EXPORT** macro and instead should create two **.def** files, one that doesn't contain **SOMInitModule** for creating the **.lib** file,

and one that does for building the DLL. Otherwise, you will run into the problem discussed earlier with duplicate symbols in the combined .lib file.

Definition of Class Hello (somdll\dyns2\os2\hello.hh):

```

1  #include <som.hh>
2
3  #ifdef WINCLIENT
4  _declspec(dllimport)
5  #endif
6  class Hello : public SOMObject {
7      #pragma SOMClassName(*, "Hello")
8      #pragma SOMIDLPass(*, "Implementation-End", \
9          "dllname = \"hello.dll\";")
10     public:
11         void sayHello() ;
12 };

```

Implementation of Class Hello (somdll\os2\dyns2\hello.cpp):

```

1  #include <iostream.h>
2  #include "hello.hh"
3
4  void Hello::sayHello()
5  {
6      cout <<"Hello world" << endl;
7  }
8
9  SOMEXTERN void EXPORT SOMLINK SOMInitModule {
10     long majorVersion, long minorVersion, string className)
11 {
12     HelloNewClass (majorVersion, minorVersion) ;
13 }

```

OS/2 Makefile (somdll\os2\dyns2\makefile):

```

1  all: hello.dll tsthello.exe
2
3  hello.dll: hello.hh hello.cpp hello.def
4      icc /Ge- -DEXPORT=_Export hello.cpp hello.def
5      implib hello.lib hello.dll
6
7  tsthello.exe: tsthello.cpp hello.lib
8      icc tsthello.cpp hello.lib
9
10 hello.def: hello.hh
11     icc hello.hh
12     sc -sir -u hello.idl
13     sc -sdef hello.idl

```

Windows Makefile (somdll\win\dyns21\makefile):

```

1 all: hello.dll tsthello.exe
2
3 hello.dll: hello.hh hello.cpp hello.def
4     icc /Ge- -DEXPORT=_Export hello.cpp hello.def
5     ilib /geni hello.def
6
7 tsthello.exe: tsthello.cpp hello.lib
8     icc -DWINCLIENT tsthello.cpp hello.lib
9
10 hello.def: hello.hh
11     icc hello.hh
12     $(SOMBASE)\bin\sc -sir -u hello.idl
13     $(SOMBASE)\bin\sc -sdef hello.idl

```

Dynamically Loading a SOM DLL

One of the more interesting features of SOM compared to native C++ is its dynamic nature. While methods can be invoked statically, as we have seen throughout the examples so far, it is also possible to locate classes, create objects, and invoke methods dynamically. By dynamically, I mean that the program does not require any information about the class prior to manipulating it. This provides a great deal of flexibility to the application programmer, as classes can be located, loaded, and manipulated on demand at run time, for example, based on user input from a screen. In addition, it also provides the capability for languages such as Smalltalk that have a dynamic model to interact with SOM and use `DirectToSOM` C++ classes, as we shall see in Chapter 9.

The following program shows how to dynamically load a class, create an object of that class, and invoke a method on it. The client program was not compiled with any information about the dynamically loaded class (the `hello.hh` header was not included in the program). The class definition uses the **SOMClassName** and **SOMNoMangling** pragmas to allow the class and method names to be easily accessed dynamically without using mangled names. I set the SOM class name to `MyHello` instead of the C++ class name `Hello` to make sure the SOM run-time was finding the file correctly based on what I specified for the **dllname** modifier. I'll discuss this a little more subsequently.

The class client provides examples of using the dynamic SOM class and method resolution facilities through `DirectToSOM`. At line 9, a pointer to the SOM Class Manager Object is retrieved by initializing the SOM environment. (See Chapter 6 for a discussion of the SOM Class Manager.) This object is then used to display the name of the DLL that contains the `MyHello` class using the **somLocateClassFile** method at lines 10 through 13.

A **somId** is created to represent the class name `MyHello` at line 10, using the **somIdFromString** function. The **somId** type is used by many of the SOM APIs to efficiently represent and manipulate string constants. **somIdFromString** is an example of a case where SOM will allocate storage using **SOMMalloc**, which must subsequently be freed using **SOMFree**. The storage for the **somId** `HelloId` is deallocated at line 24.

The **somLocateClassFile** method at line 11 looks up the class name in the interface repository, using the supplied **somId** variable. If it cannot find valid **dllname** modifier information for the class, it simply returns the class name, as illustrated at lines 15 through 18 for class `Unknown`.

At line 22, the **somFindClass** method is invoked to locate the class object for the given class, in this case `MyHello`. This method first calls the **somLocateClassFile** method to locate the DLL containing the class, and then invokes **somFindClsInFile** to return the class object. If **somLocateClassFile** failed and returned the class name, and the class name happens to match the DLL name, then **somFindClsInFile** will locate the class correctly. Otherwise, an error will be returned. I discovered this by accident: I had incorrectly specified the **dllname** modifier, but because the class name matched the DLL name, the class was still found. But when I changed the class name to something different from the DLL name, the class was not found. The reason I used `MyHello` instead of `Hello` in the example is to illustrate this distinction. If you are having trouble dynamically loading a class, you should write a small test program that displays the result of **somLocateClassFile** for the class to determine whether the DLL information is being correctly located.

At line 27, the returned class object is then used to dynamically create an instance of class `Hello` using the **somNew** method. At line 31, the **somResolveByName** method is invoked to locate the `sayHello` method. Specifying the **SOMNoMangling** pragma in the class definition allows this method to be located using `"sayHello"` instead of `"sayzhello"`. The resolved method is then invoked at line 35. Because the method is invoked as a function pointer rather than as a `DirectToSOM C++` method, the target object and **Environment** parameters must be passed explicitly.

The OS/2 version is built using the SOM 3.0 DLL initialization approach, whereas the Windows version is built using the SOM 2.x approach. Note that `-DWINCLIENT` is not necessary for compiling the client program in Windows, because the DLL is loaded dynamically, and therefore the SOM class symbols do not need to be imported from the DLL. The output for this program is:

```
MyHello dll: hello.dll
Unknown dll: Unknown
create obj dynamically
Hello world
```

Definition of Class Hello (dyncall\os2\hello.hh):

```

1  #include <som.hh>
2
3  #ifdef WINCLIENT
4  _declspec(dllimport)
5  #endif
6  class Hello : public SOMObject {
7      #pragma SOMClassName("MyHello")
8      #pragma SOMNoMangling(" ")
9      #pragma SOMIDLPass("Implementation-End", \
10                          "dllname = \"hello.dll\";")
11  public:
12      void sayHello();
13  };

```

Implementation of Class Hello (dyncall\os2\hello.cpp):

```

1  #include <iostream.h>
2  #include "hello.hh"
3
4  void Hello::sayHello()
5  {
6      cout << "Hello world" << endl;
7  }

```

Client of Class Hello (dyncall\os2\testhello.cpp):

```

1  #include <iostream.h>
2  #include <assert.h>
3  #include <som.hh>
4
5  typedef void *(*mpt)(...);
6
7  int main()
8  {
9      SOMClassMgr *cm = somEnvironmentNew();
10     somId helloId = somIdFromString("MyHello");
11     cout << "MyHello dll: " <<
12         cm->somLocateClassFile(helloId, 0, 0)
13         << endl;
14
15     somId unknownId = somIdFromString("Unknown");
16     cout << "Unknown dll: " <<
17         cm->somLocateClassFile(unknownId, 0, 0)
18         << endl;
19     SOMFree(unknownId);
20
21     SOMClass *helloClass =
22         cm->somFindClass(helloId, 0, 0);
23     assert(helloClass);

```

Continued

```

24     SOMFree(helloId);
25
26     cout << "create obj dynamically" << endl;
27     SOMObject *obj = helloClass->somNew();
28     assert(obj);
29
30     // resolve to sayHello
31     mpt mp = (mpt)somResolveByName(obj, "sayHello");
32     assert(somGetGlobalEnvironment()->_major == 0);
33
34     // invoke sayHello
35     mp(obj, somGetGlobalEnvironment());
36 }

```

OS/2 Makefile (dyncall\os2\makefile):

```

1  all: hello.i.c hello.dll tsthello.exe
2
3  hello.dll: hello.hh hello.cpp hello.i.c
4          icc /Ti+ /Ge- /B"/NOE" \
5          hello.cpp hello.i.c hello.def
6          implib hello.lib hello.dll
7
8  tsthello.exe: tsthello.cpp
9          icc /Ti+ tsthello.cpp
10
11 hello.i.c: hello.hh
12          icc hello.hh
13          sc -sir -u hello.idl
14          sc -sdef hello.idl
15          sc -simod hello.idl

```

Windows Makefile (dyncall\win\makefile):

```

1  all: hello.idl hello.dll tsthello.exe
2
3  hello.dll: hello.hh hello.cpp hello.i.cpp
4          icc /Ti+ /Ge- -DEXPORT=_Export /B"/NOE" \
5          hello.cpp hello.i.cpp hello.exp
6
7  tsthello.exe: tsthello.cpp
8          icc /Ti+ tsthello.cpp
9
10 hello.idl: hello.hh
11          icc hello.hh
12          $(SOMBASE)\bin\sc -sir -u hello.idl
13          $(SOMBASE)\bin\sc -sdef hello.idl
14          ilib /geni hello.def

```

Windows DLL Initialization File (dyncall\win\hello.i.cpp):

```

1  #include "hello.hh"
2

```



```

3  SOMEXTERN void EXPORT SOMLINK SOMInitModule(
4      long majorVersion, long minorVersion, string className)
5  {
6      MyHelloNewClass(majorVersion, minorVersion);
7  }

```

Other Programming Considerations

Default Constructor

You should always supply a default constructor for a `DirectToSOM` class. While you may not use this constructor explicitly in your application, many of the SOM frameworks, such as DSOM, require that one be present. In addition, SOM programs written using other languages typically depend upon a default constructor being available. If you are working strictly within `DirectToSOM C++`, and not using any of the frameworks, then technically you don't need to supply it. However, it's best to get in the habit, to avoid bugs later. While the SOM RRBC support makes it easy to add one if needed, run-time errors caused by a missing default constructor can sometimes be difficult to track down.

The compiler will supply a default constructor that is valid for SOM if you do not supply any constructors for the class. If you do supply constructors for the class, then a default will not be generated.

Header Files

As you have probably noticed, the `DirectToSOM C++` header files used so far all have a file extension of `.hh`. You should always put `DirectToSOM C++` classes in a file with this extension. This is not simply a convention, but is required for IDL generation, as we shall see in Chapter 7, *IDL Generation*.

Also with regard to header files, you cannot mix the `C++` bindings `.xh` header files with the `DirectToSOM C++` `.hh` header files in a single compilation unit. Each provides different definitions of classes such as `SOMObject`. You can mix programs compiled with these different headers at run time and share SOM objects between them, but you cannot mix them at compile time.

__ClassObject Member

`__ClassObject` is a static member implicitly available for all `DirectToSOM C++` classes. It can be used to retrieve the address of the corresponding class object for the class. For example, the following program invokes the `somDescendedFrom` method against the class object for `B`, passing the class object for `A`, and prints out a message indicating whether `B` is descended from `A`. Then it checks whether `A` is descended from `B`. As shown, you can retrieve the class object either through the class or the instance, since it is a static member. Note that this is not an actual member of the class—it is just treated as such by the compiler, which simply transforms a

__ClassObject expression into an expression that retrieves the class object from the **<className>ClassData** structure (see Chapter 6 for details about the layout of this structure). The output of the program is:

```
B descendent of A: true
A descendent of B: false
```

*Using the **__ClassObject** Member (classobj.cpp):*

```
1  #include <iostream.h>
2  #include <som.hh>
3
4  class A : public SOMObject {
5      #pragma SOMDefine(*)
6  };
7
8  class B : public A {
9      #pragma SOMDefine(*)
10 };
11
12 int main(int argc, char *argv[])
13 {
14     A a;
15     B b;
16
17     SOMClass *bClass = b.__ClassObject;
18     SOMClass *aClass = A.__ClassObject;
19     cout << "B descendent of A: "
20          << (bClass->somDescendedFrom(aClass)
21              ? "true" : "false") << endl;
22     cout << "A descendent of B: "
23          << (aClass->somDescendedFrom(bClass)
24              ? "true" : "false") << endl;
25 }
```

The type returned from **__ClassObject** is **SOMClass ***, and not the type of the metaclass for the class (if it differs from **SOMClass**). This is due to the possibility that the metaclass at the time the program is compiled may be different when the program is actually run, as the result of an updated class implementation, in which case the wrong type would be returned.

__isDTSClass operator

The **__isDTSClass** operator can be applied to any type and will return true or false indicating whether the type represents a DirectToSOM class. It is particularly useful within macros or template classes, where you may want to take different actions depending upon whether the class is a Direct-ToSOM class or not.

The programming example illustrates the use of this operator. The template class `Outer` is instantiated with a SOM class `A` and a non-SOM class `B` at lines 24 and 25 respectively. Then the method `showType` is invoked against each instantiation. The output of this program is:

```
This is a DTS class
This is a native class
```

Using the `_isDTSClass` Operator (`isdtscls.cpp`):

```
1  #include <iostream.h>
2  #include <som.hh>
3
4  template <class T> class Outer {
5      public:
6          void showType();
7  };
8
9  template <class T> void Outer<T>::showType()
10 {
11     if (_isDTSClass(T))
12         cout << "This is a DTS class" << endl;
13     else
14         cout << "This is a native class" << endl;
15 }
16
17 class A : public SOMObject { };
18 #pragma SOMDefine(A)
19
20 class B { };
21
22 int main(int argc, char *argv[])
23 {
24     Outer<A> a;
25     Outer<B> b;
26
27     a.showType();
28     b.showType();
29 }
```

This operator is not yet available on all platforms, (it is currently supported in Windows, but not in OS/2) but should become available with the next releases of the products.

Forward Declarations

When specifying a forward declaration of a `DirectToSOM` class, you should enclose the declaration with `#pragma SOMAsDefault(on)` and `#pragma SOMAsDefault(pop)` so that the compiler knows this is a `DirectToSOM` class; otherwise, the compiler assumes that it is a native C++ class. In general, this does

not cause problems, but it can result in error messages in certain situations where the compiler is expecting a `DirectToSOM` class, in particular when working with pointers to members.

As a simple example, the following **SOMDefine** pragma would be flagged in error without the **SOMasDefault** pragmas around the forward declaration of `A`.

```
#pragma SOMasDefault(on)
class A;
#pragma SOMasDefault(pop)

#pragma SOMDefine(A)

class A : public SOMObject {
};
```

Volatile Members

In general, the compiler-supplied methods for a `DirectToSOM` class are not volatile; in other words, they cannot be applied to volatile member functions. If you want to use volatile SOM objects, you have to define volatile versions of the member functions in order to invoke methods against those objects. Of the 10 special **SOMObject** methods, the copy constructor and assignment methods are defined to operate on a volatile object parameter, but the member functions themselves are not volatile, and thus cannot be invoked against a volatile object.

Be aware that if you do supply a volatile version of the assignment operator in particular, the compiler will not generate any other assignment methods. (See Chapter 6 for further details.) For performance reasons, you should therefore define a constant nonvolatile version of the function, otherwise all assignments will use the volatile version. For objects that are not volatile, this will cause unnecessary performance degradation.

The current release of the OS/2 and AIX VisualAge products define the attribute `_get/_set` member functions as volatile by default, but this will change in the next release of those products to be nonvolatile by default (this is already the default for Windows).

SOMObject as Private Base Class

The proposed ANSI standard dictates that the most derived class is responsible for initializing virtual base classes to ensure that the virtual base is initialized only once and in the right place. The most derived class must have access to the constructor for the virtual bases in order to invoke them, otherwise an error will occur at compile time. Because **SOMObject** is always implicitly a virtual base class, defining a class with **SOMObject** as

a private base will prevent that class from being derived from further. This is because the constructor for **SOMObject** will not be accessible to any derived classes.

For example, the following program will compile cleanly when **A** is a native C++ class, or if **A** derives publicly from **SOMObject**. However, because **SOMObject** is always a virtual base, the most derived class must construct it, so a compile-time error would occur if **A** derives privately from **SOMObject**, because **SOMObject::SOMObject()** is not accessible from **B**.

```

1  #include <iostream.h>
2  #include <som.hh>
3
4  #if SOM
5  class A : private SOMObject { };
6  #else
7  class A { };
8  #endif
9
10 class B : public A { };
11
12 B b;
```

Performance

Apart from the issues raised in the previous section, one of the major considerations when deciding to make a class a **DirectToSOM** is performance. While **DirectToSOM C++** gives you the power and flexibility of the **SOM** object model, there is an overhead in using it. For interfaces involving many calls to very simple methods, this overhead can be significant. Much the overhead is not intrinsic to the **SOM** model, and will likely be alleviated in future releases of the product.

In terms of run-time performance, every **SOM** virtual method invocation currently requires an indirection through the **SOM** class data structures to invoke the method. This is in the form of a call through a function pointer to a thunk, which is a small code sequence that performs a few setup instructions and branches to the target method. Nonvirtual and static member functions are also currently called through a function pointer. See Chapter 6, *Inside DirectToSOM C++*, for details on the calling mechanism.

In addition, instance data access, either from the client or within the implementation, currently requires a *data token* function pointer call through the **SOM** class data structures to retrieve addressability to that data. For access through the **this** pointer, one call is required within each method invocation. (For **CORBA** attributes, this is in addition to the **_get/_set** method call, if any, because the target method must still access the instance data through the data token, too. As mentioned earlier, instance

data can be made into attributes to provide external access, but can still be accessed from DirectToSOM C++ directly through the data token without calling the `_get/_set` methods.) A future release of DirectToSOM C++ will likely lessen this overhead by accessing instance data for the `this` pointer through a supplied register value. Back-end support for specifying that the result of the data token function call as invariant can also improve performance in allowing optimizations such as hoisting data token calls out of loops. Until such support is available, you can improve the performance of methods that frequently access a given data member on each invocation, perhaps inside a loop, by saving that value in a temporary and using the temporary instead.

With respect to run-time storage requirements, each SOM object contains a pointer at the beginning to the class run-time data structures. So, when a class is made into a SOM class, each object will increase in size by the size of the pointer.

In certain situations, defining a class as a SOM class can also result in an increase in the static object size. Because inline methods are currently generated out-of-line, the object size may increase noticeably over using native C++ for classes with a large number of inline methods. In addition, prolog code is currently generated for each constructor and destructor that prevents multiple base class initialization. For classes with a large number of constructors, this additional code can be discernible in the resulting object size. If you are having trouble with static object size, you may need to consider reducing the number of constructors in the class.

In most cases, however, there will be a certain set of classes that are exposed in the interface, for which SOM support is desirable. Likely there will be many more internal classes that don't require this additional functionality. DirectToSOM and native C++ classes can be mixed as the needs of the application dictate.

As with using C++ instead of C, or a high-level language instead of assembler, using DirectToSOM C++ instead of native C++ is a programmer productivity versus program efficiency trade-off. One can choose to manage aspects such as RRBC, language neutrality, persistence, and network distribution explicitly and probably produce faster code, but usually at the cost of language restrictions and increased program management, which can be both tedious and error-prone.

Common Programming Problems

This section covers a few of the most common programming problems that occur when using DirectToSOM C++, for which the solutions are not very obvious. In other chapters, including Chapter 8, *Distributed SOM*, I will cover issues specific to that topic.

Link Errors

Link errors resulting from missing symbols are among the most common errors encountered by initial users of DirectToSOM C++. These errors fall into two broad categories:

- ♦ unresolved symbols: `<className>ClassData`, `<className>CClassData`, `<className>NewClass`
- ♦ unresolved member functions and static data members

Unresolved Symbols: `<className>ClassData`, `<className>CClassData`, `<className>NewClass`

This is typically the first error encountered by the DirectToSOM C++ programmer. The three SOM class symbols (see Chapter 6, *Inside DirectToSOM C++*, for details) are exported by the class implementation according to the following rules:

1. If the class has at least one out-of-line member function, the symbols are defined and exported from the file where the definition of the first out-of-line function is encountered.
2. If the class has no out-of-line member functions, the class data structures are defined and exported from the file containing a **SOMDefine** pragma for the class.

Keeping these rules in mind, let's look at a couple of examples where the unresolved symbols error could occur. In the following program, the first out-of-line member function for class `A` is the constructor `A::A()` at line 6. Therefore, the compiler will only define and export the SOM class data structure for class `A` in the file containing the definition for `A::A()`. If the following file were compiled as-is, with no other files included, a link error would occur because the data structures for `A` would not have been defined. The way to fix this is to either include the definition for `A::A()` out-of-line in the current file or to include in the link step the file that defines that constructor.

This error is commonly encountered when implementing a class as a separate DLL or in a separate object; any client of that class must be either dynamically or statically linked to that implementation. However, it can also occur by omitting the file containing the definition of the first out-of-line member function from the link step, or not exporting the appropriate symbols from the DLL. For SOM DLLs, ensure that you are exporting the static symbols appropriately as described in the section SOM DLLs.

```

1  #pragma SOMAsDefault(on)
2
3  class A {
4      int i;
5      public:

```

Continued

```

6     A{ };
7     ~A{ };
8 };
9
10 A::~A() { };
11
12 int main(int argc, char *argv[])
13 {
14     A a;
15 };

```

In next example, class `A` has no out-of-line member functions. Therefore, the compiler will not generate the SOM class data structures for `A` until it encounters a **SOMDefine** pragma for `A`. If the file were compiled into a program as-is, it would generate unresolved symbol errors at link time because a **SOMDefine** pragma is not specified for class `A`. To fix this, either make the constructor out-of-line or add a **SOMDefine** pragma for `A`. You should not simply include the **SOMDefine** pragma in the header file, because that would result in the SOM class data structures being defined in each file that included the header file, greatly increasing the size of your program. A good approach is to include the **SOMDefine** program in a separate implementation file, and link this into your program.

```

1 #pragma SOMAsDefault(on)
2
3 class A {
4     int i;
5     public:
6     A() { i = 0; };
7 };
8
9 int main(int argc, char *argv[])
10 {
11     A a;
12 };

```

Note that **SOMDefine** is ignored by the compiler for classes that have at least one out-of-line member function. If you are encountering link errors for such classes, you must follow the instructions for the previous example. You can put as many **SOMDefine** pragmas in your program as you want, and it will have no effect.

Unresolved Symbol <className>NewClass Only

If you are getting an unresolved symbol error for <className>NewClass only, it is probably because the SOM class name does not match the name used in the **SOMInitModule** routine to initialize the class in the SOM 2.x DLL initialization approach. The DTS C++ compiler mangles SOM class names so that they are case-insensitive. This mangling may cause

a mismatch in what you would expect to use when invoking `<className>NewClass` from within the `SOMInitModule` routine. If you are using the SOM 3.0 approach and are getting this error, ensure that the class name generated in the IDL is consistent with that specified in the header file. This error may also occur if you have changed the SOM class name and did not regenerate the init/term function, or did not compile all files that use the class.

Unresolved Symbol <class>::dts__ when Compiling Template

This is another twist on the missing SOM class symbols error just discussed, but it occurs in OS/2 when the target class is a SOM template class. The missing symbols actually correspond to the SOM class data structure names `<class>ClassData`, `<class>CClassData`, and `<class>NewClass`. For example, if you dump the object `main.obj` from the *Templates* section in OS/2 with `cppfilt -b -r main.obj`, you will see that each of the SOM class symbols names is demangled to the name `zstack::dts__`, as shown in the partial output here:

```
;zstack::dts__
dts__6zstackwti_NewClass
;zstack::dts__
dts__6zstackwti_CClassData
;zstack::dts__
dts__6zstackwti_ClassData
```

The lines with the semicolons are the names that the demangler produces for the actual required symbols shown on the line directly after them. You can see that the demangler currently has trouble with template names where the SOM class symbols are appended to the end. When the linker can't find one of the symbols, it uses the demangled name to indicate the missing symbol. So, when the SOM class symbols cannot be found at link time for a template class, it results in some rather cryptic link errors. The demangler handles the symbols correctly in Windows, but not in OS/2.

When using the automatic template generation support, this error will occur if you don't compile the object files with `/Tdp`. For example, if the makefile for the template example specified just

```
icc main.obj
```

instead of:

```
icc -tdp main.obj
```

for creating the executable, these link errors would result because the template class definitions would not be compiled and linked into the executable.

Other Unresolved Symbols

As part of creating the SOM class data structures, the address of each function and static data member in the class is supplied to SOM by the DirectToSOM C++ compiler. This implies that all function and static data members must be defined by link time because there are external references to them. If you don't supply definitions for all such members, unresolved reference errors will occur at link time. This is different from native C++, where you don't need to define a member unless it is explicitly referenced in the program. If you simply turn SOM mode on for a given class, and attempt to create a library, you may discover that some methods are missing implementations that would not have mattered in native C++.

For example, the following program will compile without error as a native C++ class, because the method `foo` is not invoked. However, an unresolved symbol error for method `foo` will occur when it is compiled as a SOM class.

```

1  #ifdef SOM
2  #pragma SOMAsDefault(on)
3  #endif
4
5  class A {
6      int i;
7      public:
8          A();
9          void foo();
10 };
11
12 A::A() {}
13
14 int main(int argc, char *argv[])
15 {
16     A a;
17 };

```

Unresolved Symbols: `_ct_DTS`, `_dt_DTS`

These errors result from the problem described in the previous section, and refer to missing constructor or destructor methods for the class. In other words, the class definition contains declarations for these methods, but the method definitions were not found at link time. Because the names are a little cryptic until you are used to them, it is not always obvious what the problem is.

Link Error Checklist

The following is a checklist for problem determination when you are getting cryptic link errors when dealing with SOM classes:

- ♦ Make sure that definitions are supplied for all function and static data members of a DirectToSOM class.
- ♦ If the class has no out-of-line member functions, ensure the **SOM-Define** pragma is specified for the class exactly once in one of the files linked into the program.
- ♦ If the class has at least one out-of-line member function, ensure that this function exists and that the file containing it is linked into the program.
- ♦ When creating a DLL, make sure that the DLL exports the three SOM class symbols and that the corresponding import file for the DLL contains these symbols. In OS/2 or Windows, you can use the **ILIB** utility to display a **.lib** file's contents (in OS/2, type `ILIB name.lib`, hit Enter at the options prompt, and then type an output file name at the listing file prompt. This file will contain all the symbols contained in the **.lib** file. In Windows, type `ILIB /list name.lib`. This will generate a listing file in `name.lst`).

Exceptions

There are many reasons for getting an exception when manipulating a SOM object, but three of the more common ones are:

- ♦ *Passing an uninitialized **Environment** parameter:* Ensure that the **__SOMEnv** variable is set prior to invoking a method. This may not cause an error for all method invocations, but certain methods in the SOM run-time will explicitly check for a valid **Environment** parameter and return an exception. Also, if a method does need to indicate that an error occurred, it will get an exception if the structure is not valid. Again, future releases of the product will initialize this method implicitly, but the current releases of both OS/2 and Windows do not.
- ♦ *Corrupted method table pointer:* Each SOM object contains a pointer at the beginning to a run-time data structure, called the method table for the class, which is used by the SOM run-time to perform method resolution and access information about the object (see Chapter 6 for details). If this pointer, or the method table itself, is ever corrupted or overwritten, it will almost definitely cause an exception to occur on object manipulation.

You can verify the validity of the method table pointer in the debugger by using the following cast to view the method table:

```
(somMethodTab List *)&obj
```

Then look at the first member of this structure, `mtab`, which will be the method table for the SOM object. You should be able to visually verify that certain fields, such as `className`, are correct (refer to Chapter 6

for details about this structure). If the method table appears corrupted, you will need to determine where in your program that storage is being updated.

- ♦ *inout parameter passed as null*: This is more of a normal C++ programming problem than a DirectToSOM problem, but if you are passing a value for a parameter that is defined as **inout**, ensure that the parameter is non-null. While the method description may not indicate that a valid value is always expected, it is implicit in the definition of the **out** and **inout** parameters.

Compile-time Error when Defining Attribute Methods: “_get/_set is not a member of”

If you are defining the **_get** or **_set** member function for an attribute and are getting this message, the problem is either that the compiler is not expecting this method to be supplied or the signature you are specifying for the method does not match the one that the compiler is expecting for that method. For the former situation, make sure that you are specifying **noget** or **noset** with the **SOMAttribute** pragma for supplying a **_get** or **_set** method respectively. In the latter situation, in OS/2 or Windows, I find the easiest way to figure out exactly what the compiler is expecting is to remove the **noset** or **noget** from the attribute declaration and then compile the class implementation to a **.obj** file. Then dump the object using **CPPFILT -b -p**, which will list the exported symbols and provide the signatures for the attribute methods. This won't list the return value type, but typically the problem is in the parameter list or the method qualification. In AIX, you can generate an assembly listing using **-qlist**, and this will contain the method signature where the method is defined.

Remember that if you are using VisualAge for OS/2 version 3.0, you should compile your program with **-yxqnosomvolattr** so that the attribute methods are not volatile-qualified, as this will be the default for future releases.

Compile-time Error: “cannot be converted to”

If you are getting a “cannot be converted to” error when assigning a SOM method address to a function pointer, you have probably not defined the linkage for that function pointer correctly. On most platforms, particularly OS/2 and Windows, SOM methods are given a different linkage from that of standard methods. (In AIX, however, SOM member functions have the same linkage as other functions.) For example, in OS/2, SOM member functions have **_System** linkage, whereas the default is **_Optlink** for non-SOM functions. SOM defines the macro **SOMLINK** that provides a platform-

independent means of defining the linkage for a function as SOM when necessary. One situation where this will be necessary is in taking the address of a static member function of a SOM class.

For example, in the following program, a compile-time error will occur on the first function pointer assignment at line 8, because the function linkages are mismatched. The second assignment at line 11 is correct, because the function pointer is given the appropriate linkage.

```
1  #include <som.hh>
2
3  struct A : public SOMObject {
4      static void foo();
5  };
6
7  // incorrect linkage
8  void (*fptr1)() = &A::foo;
9
10 // correct linkage
11 void (SOMLINK *fptr2)() = &A::foo;
```


Inside DirectToSOM C++

This chapter explains how the C++ object and run-time model is mapped to that of SOM with DirectToSOM C++. The more you understand about how C++ is mapped to SOM, the easier it will be to develop and debug DirectToSOM C++ applications, in particular if you will be working with other languages or with frameworks such as DSOM. However, it is not necessary that you understand this material in order to use DirectToSOM C++, so if you are not interested in a “behind-the-scenes” view of DirectToSOM C++, you can skip this chapter.

The SOM Object Model

Figure 6.1 shows an overview of the SOM object run-time model corresponding to the program shown next. Every SOM object is an instance of some SOM class. In the figure, `obj` is a SOM object that is an instance of the SOM class `Hello`. Each SOM class is derived directly or indirectly from the common base class **SOMObject**. In this program example, the SOM class `Hello` is derived directly from **SOMObject**.

```
1 #include <iostream.h>
2 #include <som.hh>
3
4 class Hello : public SOMObject {
```

Continued

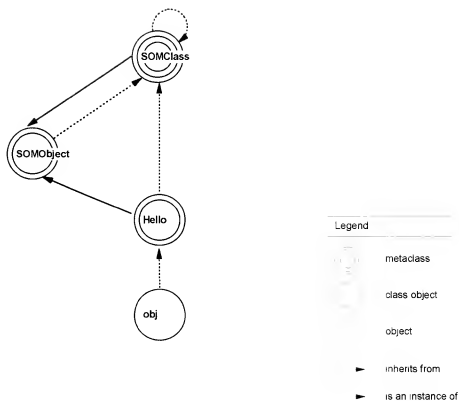


FIGURE 6.1 SOM object run-time model for class Hello.

```

5   public:
6       void sayHello();
7   };
8
9   void Hello::sayHello()
10  {
11      cout << "Hello world" << endl;
12  }
13
14  int main(int argc, char *argv[])
15  {
16      Hello obj;
17
18      obj.sayHello();
19  }

```

SOM classes also exist at run time as SOM objects. Because they are objects, SOM class objects have to be instances of some class. By default, they are instances of the *metaclass* **SOMClass**. A metaclass introduces the methods and instance data that are supported by class objects. For example,

SOMClass introduces the method **somNew**, which can be invoked against a class object to create and return a new instance of that class. In this example, invoking **somNew** against the Hello class object would return a new Hello instance. Metaclasses are used to manage and control the creation of SOM objects. You can define your own metaclass for a class object, for example, to limit the number of instances created to five. SOM also supplies several metaclasses with the Metaclass Framework. Typically, however, **SOMClass** will suffice for most applications.

The **SOMObject** class also exists as a run-time class object instance of **SOMClass**, which makes sense. Here's where the model gets a little tricky. All SOM classes are derived from **SOMObject**, so the class **SOMClass** also derives from **SOMObject**. In other words, **SOMObject** is the base class for **SOMClass**, but the run-time instantiation of **SOMObject** is an instance of **SOMClass**. Because **SOMClass** inherits from **SOMObject**, methods introduced by **SOMObject** can be invoked against the **SOMClass** class object, and because **SOMObject** is an instance of **SOMClass**, methods introduced by **SOMClass** can be invoked against the **SOMObject** class object. And finally, **SOMClass** has to be an instance of something, so it is an instance of itself. In other words, **SOMClass** is its own metaclass.

There is actually one more primitive SOM class in the SOM model that I didn't show (to keep the diagram as simple as possible). This is the SOM class **SOMClassMgr**, which keeps track of all the class objects currently created, and manages the loading and unloading of class libraries. **SOMClassMgr** inherits directly from **SOMObject**, and the run-time class object is an instance of **SOMClass**. An instance of **SOMClassMgr**, stored in the global variable **SOMClassMgrObject**, is created as part of SOM run-time initialization.

Creating SOM Objects

Like class instances in most object-oriented programming languages, SOM objects are transient. That is, SOM objects come into existence at some point during program execution and are destroyed before or during program termination. There is support available with SOM to make objects persistent, so that they endure beyond program termination, but by default they are transient. (The SOM support for persistence will be discussed in Chapter 10, *The SOMObjects Object Services* and Appendix B, *The Persistence SOM Service*.)

Creating a SOM object involves three distinct steps: storage allocation, object creation, and object initialization. Storage allocation simply involves allocating sufficient storage for the object. SOM objects are always allocated dynamically, either on the heap or on the program stack using the **alloca** function. As shown in Figure 6.2, a SOM object consists of a header portion plus the instance data for that object. The header portion points to the class method table, which contains information about

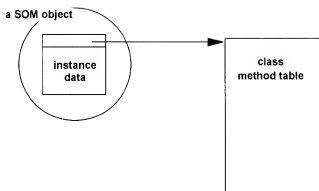


FIGURE 6.2 SOM object structure.

the class, followed by pointers to invoke methods supported by that class. I will discuss the method table in more detail later in this chapter. Thus the amount of storage required for a SOM object is the total of the instance data size for the class, plus the method table pointer (the size of a pointer).

Object creation transforms passed storage into an object upon which methods can be invoked. This is achieved by storing the method table pointer in the header portion of the object. Once this is done, the object can be used as the target of method invocations. An object is initialized by invoking a constructor, or in SOM terminology, an *initializer* method, against the object. The **SOMObject** class introduces an initializer method **somDefaultInit**, that is called by the SOM API when object initialization is required. This method is typically overridden by each subclass to provide class-specific initialization.

Each of the three steps can be performed separately. Storage allocation can be performed explicitly by the programmer, without involving the SOM run-time, but object creation and initialization must be handled through the SOM run-time. A SOM class instance is created by invoking a creation method against the corresponding SOM class object. The SOM metaclass **SOMClass** introduces a variety of class object methods for creating SOM class instances:

somNew()

Allocates storage for the object, sets the method table pointer, and initializes the object by invoking the **SOMObject** initializer method **somDefaultInit**.

somNewNoInit()

Allocates storage for the object and sets the method table pointer. Does not call **somDefaultInit**.

somRenew(void *)

Sets the method table pointer in the passed storage, sets the passed storage to all zeros, and initializes the object by invoking **somDefaultInit**.

somRenewNoInit(void *)

Sets the method table pointer in the passed storage and sets the passed storage to all zeros. Does not call **somDefaultInit**.

somRenewNoZero(void *)

Sets the method table pointer in the passed storage and initializes the object by invoking **somDefaultInit**. Does not set the passed storage to all zeros.

somRenewNoInitNoZero(void *)

Sets the method table pointer in the passed storage only. Does not set the passed storage to all zeros and does not call **somDefaultInit**.

Figure 6.3 shows the sequence of events for creating a SOM object, *obj*, of class *Hello*. In step 1, the client invokes one of the preceding methods against the corresponding *Hello* SOM class object. In step 2, the class object returns the created SOM object.

The **NoInit** versions of the preceding methods are used when the object is to be initialized with an initializer other than **somDefaultInit**; for example, one that accepts parameters. The **Renew** methods are used when you either want to perform your own storage allocation, or you have an object already allocated that you want to reuse, rather than deallocating that storage and allocating new storage. Of the **Renew** methods, the **NoZero** methods are typically used because the initializer should be responsible for initializing stor-

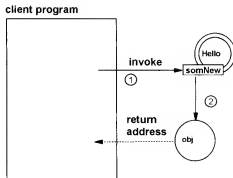


FIGURE 6.3 Creating object through *somNew*.

age. In general, using `DirectToSOM C++`, you may never invoke any of these methods explicitly. However, if you are using `DSOM` or the more dynamic facilities of `SOM`, you may use `somNewNoInit` in particular.

Contrast the previous model with that of `C++`, where storage allocation, object creation, and object initialization can be performed only as a single cohesive operation. In other words, there is no way in `C++` to allocate storage for an object, then separately make that storage into an object of a given type (other than through casting, which does not change the makeup of the underlying storage). Nor is there a way to explicitly invoke a constructor against an object. These operations are all performed at once as part of creating an object.

Creating a `DirectToSOM` Object

In order to support dynamically allocated `SOM` class instances using standard `C++` syntax, instances of `DirectToSOM` classes are implemented by the compiler as pointers to the appropriate type. In general, this representation is not visible to the programmer, as the compiler implicitly dereferences the pointer when the object is used. This approach allows a `DirectToSOM` instance to act like a standard class instance in most situations. For example, you can declare `DirectToSOM` instances with automatic or static storage duration, or allocate them dynamically, and declare them as structure members of both `DirectToSOM` and standard classes.

The `DirectToSOM` compiler implicitly uses the `SOM` model for allocating `SOM` objects. For example, consider the declaration of `obj` in the following program (where `Hello` is a `DirectToSOM` class):

```
int main()
{
    Hello obj;
}
```

`obj` appears to be a standard automatic variable, allocated on the program stack. However, the `DirectToSOM` compiler generates run-time code that essentially does the following when the declaration of `obj` is encountered:

```
Hello *obj;
obj = alloca(sizeof(Hello));
HelloClassObject->somRenewNoInitNoZero(obj);
obj->Hello::Hello();
```

The `alloca` C library function allocates storage on the program stack. This storage is passed then to `somRenewNoInitZero` for object creation. Last, the default constructor for the class is called to initialize the object. If `obj` were declared using a constructor with arguments, the appropriate constructor method would be called instead of the default constructor in the final step.

There are also some subtle differences between native and SOM class constructors. In native C++, a constructor cannot be virtual and is like a static member function, which can be called directly, passing the **this** pointer. Part of the role of a native C++ constructor is to set up the virtual function table pointers in the object. A SOM class constructor is a virtual function, and it must be called through the method resolution mechanism. A SOM constructor with the same SOM name as that of a base class constructor is treated as a virtual override of that base class method. Because a SOM constructor must be called through the method resolution mechanism, the method table must already be initialized before the constructor can be called.

Static SOM objects are allocated on the heap, for which operator **new** is used. Storage for arrays of SOM objects is not allocated as an array of pointers, rather, storage for all the array elements is allocated as a single unit. This is to support address arithmetic, so that a pointer to an array element—for example, incremented by 1—will correctly point to the next array element.

Creating SOM Class Objects

Now you know how SOM objects are created. But, creating a SOM object requires invoking a method on the corresponding class object. So the next question to answer is “Where does the class object come from?” Every SOM class implementation must support an external function call **<className>NewClass**. When this function is called, the implementation creates the SOM class object (through calls to the SOM API), and returns a pointer to this class object. If the class object has already been created, a pointer to the existing class object is returned. A DirectToSOM C++ compiler generates the **<className>NewClass** function with the SOM class implementation file (determined as discussed in Chapter 2, the *SOM Class Data Structures* section).

Figure 6.4 expands on Figure 6.3 by showing the class object creation. In step 1, the client invokes **HelloNewClass** to create the corresponding class object. In step 2, the class object is created, and returned to the client in step 3. The object is created through the **Hello** class object, as before, in step 4.

Client programs can either call **<className>NewClass** to create a required class object at program initialization time, or they can create the class object as needed at run time. Creating the class object at program initialization time may result in a class object being created that is not used, but it avoids the overhead of explicitly checking each use of the class object pointer at run time to determine if it needs to be created.

By default, the DirectToSOM compiler invokes **<className>NewClass** as part of static initialization for each class that is statically used in a program. By a statically used class, I mean implicitly by the compiler, as opposed to dynamically through a pointer to the class object retrieved from the SOM API. There is a compiler switch (**/Gz** in OS/2 and Windows) that instructs the

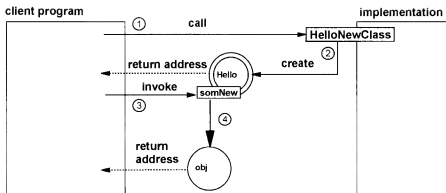


FIGURE 6.4 Creating class object through NewClass.

compiler not to statically call `<className>NewClass`, but to check each static SOM class reference and create the class object when needed.

The class implementation of `<className>NewClass` calls the SOM API function `somBuildClass`, passing information about the class in several structures. The SOM run-time constructs the class object using the supplied information and returns a pointer to that created class object, which is returned from `<className>NewClass` to the client. `somBuildClass` and the structures that it accepts are documented in the SOM `<somapi.h>` header file (see Appendix A).

Figure 6.5 expands on Figure 6.4 by showing the call to `somBuildClass`. In step 1, the client invokes `HelloNewClass` to create the corresponding class object. In step 2, the class implementation invokes the SOM function `somBuildClass` to create the target class object. In step 3, the SOM run time creates the class object, which is returned to the `HelloNewClass` function in step 4. In step 5, `HelloNewClass` returns control to the client, and returns the created class object. The `Hello` instance is then created through the `Hello` class object, as before, in step 6.

SOM Class Data Structures

Recall from Chapter 2 that the class implementation exports three symbols: `<className>NewClass`, `<className>ClassData`, and `<className>CClassData`. The implementation provides only the storage for the latter two structures, both of which are defined in `<somapi.h>`. In addition to creating the class object, `somBuildClass` also fills in these structures with class information that can then be accessed by clients.

`<className>ClassData` consists of a pointer to the class object for the class, followed by an array of *method tokens*:

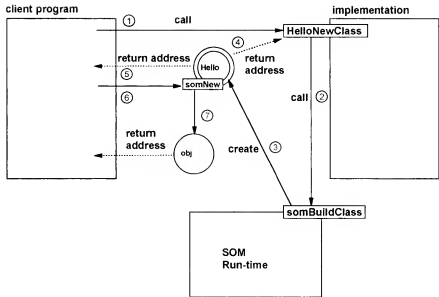


FIGURE 6.5 Creating class object with somBuildClass.

```

struct {
    SOMClass *classObject;
    somToken tokens[1];
} somClassDataStructure;
  
```

The DirectToSOM compiler uses the `classObject` member of the structure to perform any required class object manipulation. For example, in the code segment shown earlier for allocating a SOM object, the class object could be accessed as follows:

```

Hello *obj;
obj = alloca(sizeof(Hello));
HelloClassData.classObject->somRenewNoInitNoZero(obj);
obj->Hello::Hello();
  
```

If nonstatic class object initialization is requested, the compiler will generate code that checks whether the `classObject` member is NULL prior to each reference.

The tokens in **<className>ClassData** are function pointers that can be used to invoke methods on instances of that class, plus the addresses of static data members. The order of the method tokens in the **<className>ClassData** structure corresponds to the release order for the class. Every release order entry corresponds to a token array entry.

Methods may be invoked on a SOM object in several ways: *offset resolution*, *name lookup resolution*, and *dispatch-function resolution*. With offset

method resolution, the client code invokes the method through a method token found at an offset in **<className>ClassData**. The offset used corresponds to the release order location for that method, which is known at compile time. Name lookup resolution, by contrast, uses the name of the method to dynamically search for the method token. Dispatch-function resolution allows the receiving object to control how method resolution is performed. As expected, offset resolution is the most efficient means of invoking a method, because the method token is available statically, but the client code is dependent upon the location of that method token not changing, which is why the class release order cannot change.

DirectToSOM C++ (and the language bindings) use offset method resolution for method invocation. For example, given an object, `obj`, of the class `Hello` shown earlier, the method invocation of `sayHello`:

```
obj.sayHello();
```

would be translated roughly by the compiler to:

```
HelloClassData.token[0](obj, __SOMEnv)
```

Recall that DirectToSOM C++ implicitly passes the `__SOMEnv` variable as the second parameter on every SOM method invocation with callstyle **idl**.

For virtual functions, the tokens do not contain the direct address of the target method. Rather, they are pointers to *thunks*, which are small code sequences supplied by the SOM run-time that branch to the proper virtual function. For nonvirtual or static functions, the token contains the actual address of the target function. The calling sequence through the token is still the same, but no thunk is involved. Likewise, the token for a static data member contains the actual address of that data member. This allows other languages to access the data without knowing the external name.

<className>CClassData consists of a pointer to what is known as the *parent method table*, followed by a data token for the class. The remainder of the structure is opaque.

```
struct {
    somParentMtabStructPtr parentMtab;
    somIDToken             instanceDataToken;
    somMethodProc          *wrappers[1];
} somCClassDataStructure;
```

The data token is used to obtain addressability to the instance data for an object, which will be discussed in a subsequent section, *Instance Data*.

The Parent Method Table

The parent method table, shown next, provides several pieces of information used by the DirectToSOM compiler. Most important are a pointer to the

class method table (the same as that pointed to by the header portion of a SOM object), the instance data size in member `instanceSize`, and a thunk to provide direct access to the `somRenewNoInitNoZero` method.

```
struct {
    somMethodTab    *mtab;
    somMethodTabs   next;
    SOMClass        *classObject;
    somTD_somRenewNoInitNoZeroThunk somRenewNoInitNoZeroThunk;
    long            instanceSize;
    somMethodProc    **initializers;
    somMethodProc    **resolvedMTokens;
    somInitCtrl      initCtrl;
    somDestructCtrl  destructCtrl;
    somAssignCtrl    assignCtrl;
    .
    .
    .
} somParentMtabStruct;
```

`instanceSize` is used at run time whenever the size of a SOM class instance is required. The compiler uses the `somRenewNoInitNoZeroThunk` member to perform object creation, for improved efficiency over accessing the method through the class object. If the class does not supply an override of `somRenewNoInitNoZero`, the thunk will contain code to store the method table in the object directly; otherwise, the thunk will call the class-supplied method. So, using the parent method table, the storage allocation for a DirectToSOM C++ object is actually as follows:

```
Hello *obj;
obj = alloca(HelloClassData.parentMtab.instanceSize);
HelloClassData.parentMtab->somRenewNoInitNoZeroThunk(obj);
obj->Hello::Hello();
```

The parent method table members `initializers` and `resolveMTokens` are not used by the DirectToSOM C++ compiler. The three members `initCtrl`, `destructCtrl`, and `assignCtrl` are used for controlling base class initialization, destruction, and assignment. I will discuss their use later in this chapter in the *Inheritance* section.

The Method Table

The method table is defined by the following structure:

```
struct somMethodTabStruct {
    SOMClass        *classObject;
    somClassInfo    *classInfo;
    char            *className;
    long            instanceSize;
```

Continued

```

        long          dataAlignment;
        long          mtabSize;
        long          protectedDataOffset;
        somDToken      protectedDataToken;
        somEmbeddedObjStruct *embeddedObjs;
        /* remaining structure is opaque */
        somMethodProc* entries[1];
    } somMethodTab;

```

The first member of the method table, `classObject`, provides addressability to the SOM class object for the given class. `classInfo` is an opaque structure containing class information that is used by the SOM run time. `className` is the SOM name of the class. `instanceSize` provides the same information, the instance data size, as does the member in the parent method table, while `dataAlignment` provides the alignment for that data. `protectedDataOffset` and `protectedDataToken` are used for accessing protected and private instance data, which I will discuss in a subsequent section, *Instance Data*. `embeddedObjs` is used to manage SOM objects that are embedded in the current class, that is, data members that are themselves SOM objects. The remainder of the structure provides addressability to the actual class methods, and is accessed only through the SOM run time.

To give you a better understanding of how the various structures fit together, Figure 6.6 illustrates the interconnections between the run-time components for the following SOM class:

```

1  class A : public SOMObject {
2      int a1;
3      static int a2;
4      void foo();
5  public:
6      int a3;
7      int a4;
8      virtual void bar();
9      virtual void bar(int);
10 private:
11     int a5;
12 };

```

Inheritance

The array of method tokens in the **ClassData** structure consists only of those methods introduced by that class. For most member types, a declaration in a class constitutes an introduction in that class. But, for virtual function members, the introducing class is the first base class to declare that member. Thus, for virtual function members that are inherited from a base class, there is no entry in the method token array. When an inherited virtual function is invoked, the corresponding method token in the **ClassData** structure for the base class is used. For example, consider the following class B, which inherits from A shown previously:

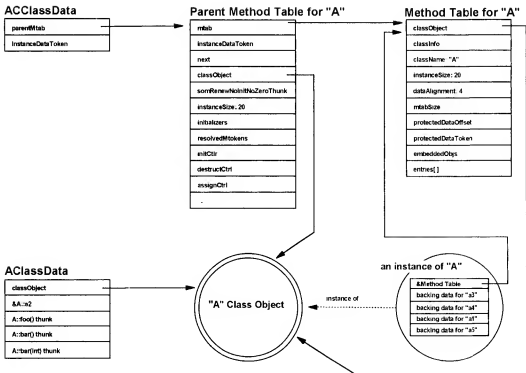


FIGURE 6.6 SOM class data structures.

```

1 class B : public A {
2     int b1;
3     void foo();
4     void foo2();
5 public:
6     int b2;
7     void bar();
8     void bar(int);
9 };

```

The release order for `B` consists of methods `foo()` and `foo2()` which are both nonvirtual functions. `bar()` and `bar(int)` are virtual functions that are introduced in class `A`, so are not part of the release order for class `B`.

Given an object `objB` of class `B`, `objB.foo()` would translate to:

```
BClassData.token[0](objB, __SOMEnv)
```

But `objB.bar()` would translate to:

```
AClassData.token[2](objB, __SOMEnv)
```

SOM uses the method table of the instance passed, in this case `objB`, to determine the method to call. Each version of an overridden method within a class hierarchy will appear at the same location in each class' method table entries array. For example, if `A::bar()` occurred at index 37 in the method table for class `A`, then `B::bar()` would also appear at index 37 in the method table for class `B`. (These locations bear no relationship to the release order for method tokens.)

So for single inheritance, the method token thunk can simply use the method index to find the target method in the entries array. Depending upon the class of the instance passed, the appropriate method will be correctly invoked from its corresponding method table using the method index. For multiple inheritance, the thunk may contain a formula to determine the method index to use.

The preceding model is one of the reasons that thunks are used for method tokens rather than directly accessing the class method. The SOM run-time can adjust the location of the methods and update the thunks with new index values as additional classes are added. While the actual algorithms used are opaque and can change over time, they are based on graph-coloring algorithms such as those described in Hamilton (1996a).

Another aspect of inheritance is the three members `initCtrl`, `destructCtrl`, and `assignCtrl` in the parent method table. The SOM run-time fills them in when a class is created. The DirectToSOM compiler (and the language bindings) use these members to prevent multiple base class initialization, destruction, and assignment. Every SOM initialization, destruction, and assignment method is also responsible for invoking the corresponding base class method for all of its base classes. Such methods accept a hidden parameter, which is a pointer to the appropriate control structure. Prior to a method invoking a corresponding base class method, it checks the control structure to determine whether that base class has been handled. If not, it invokes the method, passing the control structure, and updates the control structure.

Because a particular method can be invoked either from a derived class method or directly from the user program, it is necessary to have a mechanism to set up the control structure correctly for each possible situation. This is what the members of the parent method table are used for. When an initialization, destruction, or assignment method is invoked directly from the program, the compiler implicitly passes null for the control structure value. When null is received, the method will copy the control structure from its parent method table and use this to handle base classes; otherwise, it will use whatever is passed to it.

Instance Data

As discussed in Chapter 3, instance data in a SOM class is reordered into contiguous chunks by access. This reordering provides efficient access to

data members, while enabling RRBC. The storage is allocated as a single piece, but is addressable as two pieces: public data is grouped into one piece, and protected and private data are grouped together, with private data following the protected data.

When a DirectToSOM program requires direct access to class instance data, it calls a data token, which returns the instance data address. For accessing public instance data, the `instanceDataToken` member of the **<className>ClassData** structure is called, passing the instance variable. This returns the address of the public data for the class. The offset within the access group for a particular data member is statically compiled into the accessing code and is added to the data token result to obtain the address of the particular member. For example, the instance data order for class A shown earlier is:

```
public from A:  a3
                a4

private from A: a1
                a5
```

To obtain addressability to the member `a4` within instance `obj` of class A, the following code sequence would be used on systems with an integer size of 4:

```
ACClassData->instanceDataToken(obj) + 4
```

For accessing protected instance data, either `protectedDataOffset` in the method table can be added to the `instanceDataToken` result, or `protectedDataToken` can be called to return the address directly. The DirectToSOM C++ compiler currently uses `protectedDataToken`. Private data is accessed from the protected data address by adding the size of the protected data. This implementation requires that all code that accesses private data members be recompiled if a new protected data member is added to the class.

For example, the following code sequence would be used to obtain the address of member `a5` within `obj`:

```
ACClassData->mtab->protectedDataToken(obj) + 4
```

In addition to grouping the instance data separately by access specifier, the storage for each class is also grouped individually. Given an instance `obj2` of the class B shown earlier, the instance data member `b2` would be accessed using `BClassData`:

```
BClassData->instanceDataToken(obj2)
```

but the data member `a1`, inherited from A, would be accessed using `ACClassData`:

```
ACClassData->instanceDataToken(obj2)
```

Separating data access by class allows an additional base class to be introduced without affecting existing client code, as there is no dependency upon the location or size of base class data.

SOMObject Methods

The **SOMObject** class (see Appendix A) introduces many methods, 10 of which have special significance with respect to DirectToSOM C++ classes. These methods are:

```
void    somDefaultInit(somInitCtrl* ctrl);
void    somDestruct(octet doFree, somDestructCtrl* ctrl)

void    somDefaultCopyInit(somInitCtrl* ctrl, SOMObject* fromObj)
void    somDefaultConstCopyInit(somInitCtrl* ctrl, SOMObject* fromObj)
void    somDefaultVCopyInit(somInitCtrl* ctrl, SOMObject* fromObj)
void    somDefaultConstVCopyInit(somInitCtrl* ctrl, SOMObject* fromObj)

SOMObject* somDefaultAssign(somAssignCtrl* ctrl, SOMObject* fromObj)
SOMObject* somDefaultConstAssign(somAssignCtrl* ctrl, SOMObject* fromObj)
SOMObject* somDefaultVAssign(somAssignCtrl* ctrl, SOMObject* fromObj)
SOMObject* somDefaultConstVAssign(somAssignCtrl* ctrl, SOMObject* fromObj)
```

These methods are significant because the DirectToSom C++ compiler implicitly maps corresponding C++ class methods to these SOMObject methods as follows for a given class X:

X()	somDefaultInit
~X()	somDestruct
X(X&)	somDefaultCopyInit
X(X const &)	somDefaultConstCopyInit
X(X volatile &)	somDefaultVCopyInit
X(X const volatile &)	somDefaultConstVCopyInit
operator=(X&)	somDefaultAssign
operator=(X const &)	somDefaultConstAssign
operator=(X volatile &)	somDefaultVAssign
operator=(X const volatile &)	somDefaultConstVAssign

All 10 methods are treated as virtual by the compiler (even though you can't declare virtual constructors in C++). This implies that these methods are considered to be introduced by **SOMObject** and cannot be in the release order for the current class. In addition, when one of these methods is invoked against an object, the **SOMObjectClassData** structure is used to access the method token.

Note that each of the 10 methods accepts a control structure as the first argument, as discussed earlier. When the compiler translates a method invocation into a SOMObject method invocation, it will implicitly pass null

for the control argument. **somDestruct** actually accepts a control structure as the second argument, the first argument being a flag indicating whether the underlying storage should be freed by the destructor.

When a DirectToSOM object is created with the default constructor, the compiler initializes the object by invoking **SOMObject::somDefaultInit**, which is at offset 24 in the **SOMObjectClassData** structure. The object creation code shown at the beginning of the chapter thus becomes:

```
Hello *obj;
obj = alloca(HelloClassData.parentMtab.instanceSize);
HelloClassData.parentMtab->somRenewNoInitNoZeroThunk(obj);
SOMObjectClassData.tokens[23](obj, NULL);
```

Note that because the method **somDefaultInit** has callstyle **OIDL** (as discussed in Chapter 5), an **Environment** parameter is not passed to the method.

SOM Constructor and Destructor Methods

The mapping for the SOM constructor and destructor methods is fairly straightforward. The compiler will supply constructor methods, following C++ rules, if the programmer does not supply any. If a destructor is not supplied, one is generated that maps to **somDestruct**. The destructor will have no body, and will only be generated in order to ensure that base class destructors are invoked.

If the programmer does supply any of the constructor or destructor methods that map to the 10 special **SOMObject** methods, the DirectToSOM compiler simply generates a method with the appropriate SOM signature, instead of the standard C++ constructor/destructor method. The compiler will generate prolog code to handle the control structures and ensure that corresponding base classes methods are called only once.

SOM Assignment Methods

The mapping for assignment methods is a little trickier than for constructors and destructors, because you can take the address of C++ **operator=** methods. So, if the programmer supplies an **operator=** method, the compiler cannot simply generate the method with a different underlying signature. Instead, the compiler must generate SOM assignment methods that map to the **operator=** methods.

If the programmer does not supply an **operator=** method, then the compiler generates the methods required by C++ semantics and SOM assignment methods with appropriate bodies and signatures, as shown previously. Each will accept and manage the control structure to ensure that base class assignment occurs only once. Invocations of either **operator=** or

the SOM assignment methods will result on these generated methods being called.

If the programmer does supply an **operator=** method, the compiler generates SOM assignment methods that invoke the **operator=** method where possible. Calls using **operator=** will directly invoke the programmer-supplied methods, whereas calls using the SOM method names will invoke (through the generated compiler method) the supplied **operator=** method, where an appropriate method is available. For example, if only a non-constant **operator=** is supplied, then invocations of either **operator=** or **somDefaultConstAssign** with a constant object will not have an appropriate method available.

Note, however, that the assignment control vector is not supported for programmer-supplied **operator=** methods. For this reason, if you explicitly supply an assignment method for a SOM class, you should supply the method using the SOM name, in which case the compiler will implicitly generate prolog code to handle the control structure, as for constructors and destructors. As with the compiler-generated SOM assignment methods, invocations of **operator=** will call the programmer-supplied SOM assignment method. For example, in the following program, both method invocation on lines 23 and 24 will invoke the supplied **somDefaultConstAssign** method.

```

1  #include <som.hh>
2  #include <iostream.h>
3
4  class Hello : public SOMObject {
5  public:
6      SOMObject *somDefaultConstAssign(
7          somAssignCtrl *, const SOMObject *);
8  };
9
10 SOMObject *Hello::somDefaultConstAssign(
11     somAssignCtrl *ctrl, const SOMObject *fromSOM)
12 {
13     Hello &from = (Hello &)(*fromSOM);
14     cout << "somDefaultConstAssign" << endl;
15     return this;
16 }
17
18 int main(int argc, char *argv[])
19 {
20     Hello obj;
21     const Hello obj2;
22
23     obj = obj2;
24     obj.somDefaultConstAssign(NULL, &obj2);
25 }
```

Because the signatures of the SOM assignment methods defined in **SOMObject** specify **SOMObject *** for the source object parameter and the

return value, overriding the method as shown is not as natural as one would like. The Metaware DirectToSOM C++ compiler provides an alternative means for overriding the SOM assignment methods, using the special method **somAssign**. The **somAssign** method is written exactly as you would write an **operator=**, just with a different name, for more natural programming. For example, the **somDefaultConstAssign** method in the previous example would be written using **somAssign** as shown in the programming example that follows. The control vector is implicitly passed by the compiler. It is expected that the VisualAge C++ product family will adopt this support in future releases.

```

1  #include <som.hh>
2  #include <iostream.h>
3
4  class Hello : public SOMObject {
5  public:
6      Hello& somAssign(const Hello&);
7  };
8
9  Hello& Hello::somAssign(const Hello &from)
10 {
11     cout << "somAssign" << endl;
12     return *this;
13 }
14
15 int main(int argc, char *argv[])
16 {
17     Hello obj;
18     const Hello obj2;
19
20     obj = obj2;
21     obj.somAssign(obj2);
22 }
```

Name Mangling

SOM is case-insensitive, so all names presented to it must be unique without respect to case. In particular, class names cannot differ only by case. In order to ensure that unique DirectToSOM C++ names are also unique in SOM, the class names are subject to a case-insensitive conversion:

- ♦ Uppercase letters are converted to the lowercase equivalent, prepended by lowercase *z*.
- ♦ *z_* is used to mean lowercase *z*.

Thus *Hello* becomes *zhello* and *ZebraClassZz* becomes *zzebrazclasszzz_*.

This converted name is known as the SOM class name, as opposed to the C++ class name. In other words, *zhello* is the SOM class name for the

C++ class named `Hello`. The SOM class name is used in generating the SOM class data structures; so, for example, the final form of the object storage allocation code shown earlier is:

```
obj = alloca(zhelloClassData.parentMtab.instanceSize);
zhelloClassData.parentMtab->somRenewNoInitNoZeroThunk(obj);
SOMObjectClassData.tokens[23](obj, NULL);
```

You can use the **SOMClassName** pragma to assign a specific SOM name for a class, which is why the **ClassData** for **SOMObject** does not appear converted above. In the header file `<somobj.hh>`, **SOMObject** is given a class name of **SOMObject**. Otherwise, the name would be converted to `zszozmzobject`.

All access to members of a **DirectToSOM** C++ class is done through the SOM class data structures, so for the most part, the underlying name of the data member is irrelevant. The one situation where SOM does take into account the member name is in determining method overriding. SOM does not support name overloading, though, so overrides are based solely on the method name, not its signature. This implies that method names must be mangled, and in a standard way for all **DirectToSOM** C++ compilers, in order that method overriding can work across different **DirectToSOM** C++ compilers. The mangled name is known as the SOM name for the method.

The **DirectToSOM** C++ name-mangling scheme is based loosely on that of Cfront [Ellis, 1990], with the additional requirement that the names produced must be case-insensitive. Names are mangled first, and then converted for case-insensitivity. For example, in the following class:

```
1 #include <som.hh>
2
3 class SomeClass : public SOMObject {
4 public:
5     int fooBar(SomeClass *);
6     void fooBar(int);
7     char *fooBar(int, int);
8 };
```

the three methods are mangled to the SOM method names `foozbar__fpzsomezclass`, `foozbar__fi`, and `foozbar__fii`.

If you will be using a class from a language other than **DirectToSOM** C++, you will probably want to use the **SOMNoMangling** pragma to turn name mangling off for method names. But this will not handle name overloading, so you would also need to use the **SOMMethodName** pragma to specifically assign method names to those methods with overloaded names.

SOM names are provided to SOM in the data structures passed to the **somBuildClass** function for registration with the SOM run time. These names need not match the underlying method names that are exported from the object module, because the external names are not used to access the

method—all method access is done indirectly through the SOM class data structures. The physical function name is not used at all by SOM (except that its address is supplied to the SOM run-time as part of the **somBuildClass** structures so SOM can store its address in order to invoke it later).

The VisualAge C++ product uses the standard C++ mangling scheme and adds **__DTS** to all SOM method names to allow a DirectToSOM and non-DirectToSOM version of the same class to coexist. For example, the exported names for the preceding three methods are: `fooBar__DTS__9SomeClassFP9SomeClass`, `fooBar__DTS__9SomeClassFiT1`, and `fooBar__DTS__9SomeClassFi`. (You can display them using the **CPPFILT** command in OS/2 or Windows. Note that the **CPPFILT** demangler does not remove the **__DTS** from the SOM method names.)

IDL Generation

You will need to generate IDL to share DirectToSOM C++ classes with programs written using other languages, such as the generic C++ language bindings or Smalltalk, or if you will be using any frameworks or services such as DSOM that depend upon class information contained in the interface repository. In addition, looking at the generated IDL can help in problem determination, as it clarifies how the DirectToSOM compiler is mapping a given class to SOM. This chapter explains how to generate IDL from C++ and programming considerations for doing so.

Generating an IDL File

The DirectToSOM C++ compiler generates IDL only from files with an **.hh** extension. If the file does not have an **.hh** extension, IDL will not be generated. In OS/2 and Windows, the **/Fs** command line options controls IDL generation. The format of the **/Fs** option is:

```
/Fs [+ | - | filename | directory ]
```

The default is **/Fs+**, which generates a corresponding **.idl** file for every **.hh** file that is specified on the command line. (The **/Fs** option only applies to **.hh** files; any other files specified on the command line are processed as

normal.) The IDL file name will be the **.hh** file name with an extension of **.idl**. For example, the following command will generate two **.idl** files, **hello.idl** and **hello2.idl** (assuming **hello.hh** and **hello2.hh** exist and there are no compilation errors):

```
icc hello.hh hello2.hh
```

The IDL generation is the output of the compilation for each file. In other words, no object or executable will be created. The compiler does not generate an object file for an **.hh** file, regardless of the **/Fs** setting.

You can specify a file name with the **/Fs** option to control the output IDL file name. If you specify a file name with no extension, the default extension will be **.idl**. For example, the following generates two IDL files, **name1.idl** and **name2.out**, in the current directory:

```
icc /fsname1 hello.hh /fsname2.out hello2.hh
```

IDL will not be generated implicitly for any **.hh** files that follow one with an **/Fs** file name option, as **/Fs** with a file name essentially changes the default of **/Fs+** to **/Fs** file name, which sets it to **/Fs-** for the remaining files. To generate IDL for the remaining files, you must explicitly specify **/Fs+** to turn the default back on (this will apply to any remaining files—you only need to specify it once), or specify **/Fs** with a file name for each subsequent **.hh** file. For example:

```
icc /fsname1 hello.hh hello2.hh
```

generates only one **.idl** file for **hello.hh** in the file **name1.idl**, whereas:

```
icc /fsname1 hello.hh /fs hello2.hh
```

will generate two files, **name1.idl** and **hello2.idl**.

Finally, you can control the output file directory by specifying a directory name with **/Fs**. This will apply to all **.hh** files listed, not just the first. For example, the following generates the IDL files **hello.idl** and **hello2.idl** into the directory **\jennifer\tmp**:

```
icc /fsh:\jennifer\tmp\ hello.hh hello2.hh
```

Note that you must specify a final 'slash' after the directory name. On other platforms, the option for handling IDL generation may be different. See your compiler documentation for details.

Before examining the IDL generated for a **DirectToSOM C++** class, I will provide a brief overview of IDL to explain the basic concepts as they apply to **DirectToSOM C++**. If you want to know more about IDL, refer to the **SOMObjects Toolkit** documentation. If you have not worked with IDL

before, you will find it is similar to C++, with some extensions. The IDL used by SOM conforms to the CORBA standard IDL language.

An Overview of IDL

IDL is processed by the SOM compiler to produce a variety of outputs, such as language bindings, and to update the interface repository, which I will discuss later in this chapter. The SOM compiler is invoked with the command **sc**, specifying the IDL file(s) to process, along with any optional arguments.

In IDL, the keyword **interface** is used to denote a class. As with DirectToSOM C++ classes, all IDL classes inherit from **SOMObject**. An IDL class consists basically of two parts, an interface section and an optional implementation statement, as follows:

```
interface class-name : parent-class1 [, parent-class2, ...]
{
    constant declarations      (optional)
    type declarations          (optional)
    exception declarations     (optional)
    attribute declarations     (optional)
    method declarations        (optional)
    implementation statement   (optional)
};
```

The *interface* declarations specify any new methods, constants, and types that the class exports. They can appear in any order within the interface. At least one parent class must always be specified for an IDL class, which may be simply **SOMObject** or some other IDL class. IDL describes only the interface to a class, and not its implementation, which is quite different from C++, where all information about the class must be specified in the class definition.

The *implementation* statement is specific to SOM IDL and is not part of the CORBA standard. It provides details, such as method overriding information, about how the class will be implemented. A class can have only one implementation statement, with the following structure:

```
#ifdef __SOMIDL__
implementation
{
    instance declarations      (optional)
    passthru statements        (optional)
    modifier statements        (optional)
};
#endif
```

Because the implementation statement is specific to SOM IDL, it should be guarded by an `#ifdef __SOMIDL__` directive.

Instance variables in the implementation statement are declared in IDL the same way as in C++, with a type name followed by the instance variable name. SOM doesn't support instance variables as part of the class interface, unless they are attributes. **passthru** statements are used to generate arbitrary code into specific language bindings files. DirectToSOM C++ does not currently generate any **passthru** statements and, in general, their use is not recommended.

The bulk of the IDL generated by the DirectToSOM C++ compiler for a class consists of modifier statements in the implementation statement. There are two types of modifier statements: an unqualified modifier, which applies to the entire interface:

```
modifier;
modifier = value;
```

and a qualified modifier:

```
qualifier : modifier;
qualifier : modifier = value;
#pragma modifier qualifier : modifier;
#pragma modifier qualifier : modifier = value;
```

where the qualifier is an IDL definition or user-defined name. An example of an unqualified modifier is:

```
align=4;
```

which specifies an alignment of 4 for a class. An example of a qualified modifier is:

```
anInteger: length=4;
```

which indicates that the instance variable `anInteger` is 4 bytes in size. If there are multiple modifiers for a given qualifier, they can be listed together, separated by a comma. Many of the modifiers that will be discussed in this chapter apply only to DirectToSOM C++, and thus are not documented in the SOMObjects Toolkit IDL documentation.

The Generated IDL

Now that we know a little bit about IDL and how to generate an IDL file, let's take a look at what we get when we generate an IDL file. The example shown next illustrates the IDL generated for the DirectToSOM class `Hello`. Note that many of the modifiers shown are not documented in the SOMObjects documentation, because they apply only to IDL files that are

generated by a DirectToSOM C++ compiler. (Note that if you generate the IDL on your system, you may find the result slightly different from what I show here due to modifications to IDL generation in the compiler. Of course, any changes would be done in an upwardly compatible way, but there is currently, for example, some redundant information being generated in the IDL file that may be removed in future releases. In addition, there are differences between the IDL generated for different platforms.) In general, you won't need to be concerned with all the details of IDL generation, but this will help you understand basically what is being generated. The IDL file in this example was produced with this command:

```
icc -c hello.hh
```

Lines 1 through 8 in the IDL file are fairly straightforward—you will find this type of header information in a typical C header file. At line 9, the IDL file `<somobj.idl>` is included. This is the IDL counterpart to the `<somobj.hh>` header file that is included by `<som.hh>`. (As discussed in Chapter 2, `<somobj.hh>` is generated from `<somobj.idl>`.) The remainder of the file contains the IDL class definition.

Line 10 is a forward declaration for `zhello`, while lines 11 through 45 form the definition of, the IDL class `zhello` corresponding to the DirectToSOM C++ class `Hello`. Note that the C++ name `Hello` is mangled to the SOM name `zhello` according to the rules described in Chapter 6. In this IDL file, the forward declaration is not necessary, but the compiler generates one by default for each IDL class prior to generating the bodies, to handle references between classes.

The class `zhello` inherits explicitly from **SOMObject**, using the same syntax as with C++. As with DirectToSOM C++ classes, all IDL classes inherit ultimately from **SOMObject**. SOM doesn't have a concept of access specifiers for inheritance (**public**, **protected**, **private**) like C++ does, so base classes are always public in IDL.

In this example, the interface section of the class has a single method declaration at line 12 for the method `sayzhello__fv`. This is the SOM name corresponding to the method name `foo` in the DirectToSOM C++ class. The bulk of the class information is contained in the implementation statement, from lines 14 through 43, which is guarded by an `#ifdef __SOMIDL__` at line 13.

The first line of the implementation statement, at line 15, gives the alignment for the class, which is 4, based on the integer data member `anInteger`. Following on line 16 is the declaration of the instance variable `anzinteger`, the mangled representation of `anInteger`. Lines 17 and 18 supply a qualified modifier for this variable. The **cxxmap** modifier indicates the original C++ name of the variable, while the **cxxdecl** modifier at the end of line 18 supplies the original C++ declaration. **cxxmap** is sup-

plied in addition to **cxxdecl** to allow the SOM **hh** emitter to generate a **SOMDataName** pragma containing the C++ name and the SOM name without having to parse **cxxdecl**. These two modifiers have no effect on the IDL class definition itself. They are used only for supplying information specific to DirectToSOM C++, which the **hh** emitter uses when generating an **.hh** file from an IDL file. (I will discuss the **hh** emitter at the end of this chapter.)

nonstaticaccessors applies only to attributes (which will be discussed later in this chapter), and has no effect for this instance variable. **nonstaticaccessors** should not be generated for nonattribute data members (this will likely be fixed in the next release of the products). **offset** indicates the location within the instance data where the member appears; in this case, it is the first data member in the class. **align** and **size** supply the alignment and size in bytes of the member respectively. Last, **public** indicates that the instance variable has public access in the DirectToSOM C++ class.

Lines 19 and 20 form a qualified modifier for the method `sayzhello_fv`. All the modifiers in this particular list actually have no effect on the IDL definitions itself. They are used only for supplying information specific to DirectToSOM C++, which the SOM emitter will use to regenerate an **.hh** file from the IDL file. Most have already been explained, and have the same meaning. **nonstatic** indicates that the C++ method is neither static nor virtual. (I will discuss the mapping from C++ methods to SOM methods later in this chapter.)

Lines 21 through 30 contain overrides of the 10 special SOM methods. Recall from Chapter 6 that the DirectToSOM compiler will generate method bodies for each of these methods if none is supplied. The **public** modifiers given for these methods do not affect the IDL definition, but the **override** modifier is relevant. It indicates to the SOM compiler that the class overrides each of these methods (the **override** modifier will be discussed in more detail later in this chapter).

The **declarationorder** modifier, lines 31 through 34, is also supplied solely for the purpose of generating an **.hh** file. It specifies the order in which the various pieces should be regenerated in the C++ source. The SOM methods actually don't need to appear in this list. (The reason that only five appear is that these are the methods that are actually generated. The rest are mapped to these methods. For instance, **somDefaultAssign** is mapped to **somDefaultConstAssign**.)

Line 35 through 37 contain the **releaseorder** modifier for the class. If this is not specified, a warning message will occur when compiling the IDL. As with DirectToSOM C++, every method introduced by the class must appear in the release order. In addition, instance data members appear at the end of the release order. The order of instance data members in the release order determines their order in the language bindings data structures generated by the SOM compiler.

Line 38 indicates the class is a newer CORBA-compliant class, for which the **Environment** parameter is passed on every method call. Earlier versions of SOM did not support the **Environment** parameter, and for such classes a **callstyle** of **oidl** would be specified (see Chapter 5).

Line 39 indicates that the class definition originally came from DirectToSOM C++, and that the class is implemented using DirectToSOM C++. SOM will assume the DirectToSOM C++ defaults rather than the SOM defaults for the class definition. Line 40 specifies which base classes should be directly initialized by this class. If this modifier is not specified, the SOM compiler will assume a default of all the direct bases classes. This information is used in constructing the initialization control structure discussed in Chapter 6.

Lines 41 and 42 contain **cxxmap** and **cxxdecl** modifiers with similar functions as that of the method and instance variables qualifiers. They too are used only by the **hh** emitter.

DirectToSOM C++ Class Hello (file hello.hh):

```

1  #include <som.hh>
2
3
4  class Hello : public SOMObject {
5      public:
6          int anInteger;
7          void sayHello();
8  };
9
```

Generated IDL for Class Hello (file hello.idl):

```

1  #ifndef __hello_idl
2  #define __hello_idl
3  /*
4   *
5   * Generated on Sat Mar 23 16:54:45 1996
6   * Generated from hello.hh
7   * Using IBM VisualAge C++ for OS/2, Version 3
8   */
9  #include <somobj.idl>
10 interface zhello;
11 interface zhello : SOMObject {
12     void sayzhello__fv ();
13 #ifdef __SOMIDL__
14     implementation {
15         align=4;
16         long anzinteger;
17         anzinteger: cxxmap="anInteger",offset=0,align=4,size=4,
18                     nonstaticaccessors,public,cxxdecl="int anInteger;";
19         sayzhello__fv: public,nonstatic,cxxmap="sayHello()",
20                     cxxdecl="void sayHello();";

```

Continued

```

21     somDefaultConstVAssign: public, override;
22     somDefaultConstAssign: public, override;
23     somDefaultConstVCopyInit: public, override, init;
24     somDefaultInit: public, override, init;
25     somDestruct: public, override;
26     somDefaultCopyInit: public, override;
27     somDefaultConstCopyInit: public, override;
28     somDefaultVCopyInit: public, override;
29     somDefaultAssign: public, override;
30     somDefaultVAssign: public, override;
31     declarationorder = "anzinteger, sayzhello__fv,
32         somDefaultConstVAssign, somDefaultConstAssign,
33         somDefaultConstVCopyInit, somDefaultInit,
34         somDestruct";
35     releaseorder:
36         sayzhello__fv,
37         anzinteger;
38     callstyle = idl;
39     dtsclass;
40     directinitclasses = "SOMObject";
41     cxxmap = "Hello";
42     cxxdecl = 'class Hello : public virtual SOMObject';
43 };
44 #endif
45 };
46 #endif /* __hello_idl */

```

Mapping DirectToSOM C++ Classes to IDL

Name Mangling

If you are generating IDL to be used from other languages, you will most likely want to prevent the compiler from generating mangled names, as the mangled names are not very easy or natural to use from other languages. There are four pragmas that you can use to effect name mangling: **SOMNoMangling**, **SOMMethodName**, **SOMDataName**, and **SOMClassName**. (See Chapter 4 for details on the syntax of these and other pragmas mentioned in this chapter.)

SOMNoMangling turns off name mangling and case-insensitive conversion for function and data member names, so that the SOM name for each member will be the same as the C++ name. Recall that SOM does not support method name overloading by parameter types. If you do have overloaded method names in a DirectToSOM C++ class, specifying **SOMNoMangling** will result in compile-time errors for each overload, indicating that the SOM method name has already been used for that class. For such situations, you must use the **SOMMethodName** to explicitly provide a different name for each overloaded method. In addition, if you have data members names that differ only by case, you will need to use the **SOMDataName** pragma to assign unique names.

The **SOMNoMangling** pragma applies only to class data member names. Specifying **SOMNoMangling** will not prevent case-insensitive conversion of class names (although a future release of the produce may support this). In order to produce an unmangled SOM class for a class, you must use the **SOMClassName** pragma, specifying the desired SOM class name.

As an example, the following shows the use of these pragmas with the class `Hello`, which has been updated to add an overload of method `sayHello()`. At line 4 in the C++ file, the **SOMNoMangling** pragma is specified for the class, which turns member name mangling off for that class. At line 5, **SOMClassName** is used to indicate that the SOM name for the current class should be `Hello`. Finally, the method `sayHello(char *)` is given a SOM name of `sayHello_string` using the **SOMMethodName** pragma at line 10. This is necessary because, due to the **SOMNoMangling** pragma, the default SOM method name of `sayHello` has already been assigned to the method `sayHello()` at line 8.

These changes to the generated names make the IDL file a little easier to read. The SOM class name of `Hello` is used at lines 10 and 11, instead of `zhello` in the earlier example. The SOM method name for the C++ method `sayHello()` is `sayHello`, at line 12, and that of `sayHello(char *)` is `sayHello_string`, at line 13. In addition, the SOM name for data member `anInteger` is `anInteger`, at line 18, instead of `inzinteger` in the previous example.

DirectToSOM C++ Class Hello (file `hellow.hh`):

```

1  #include <som.hh>
2
3  class Hello : public SOMObject {
4      #pragma SOMNoMangling(*)
5      #pragma SOMClassName(*, "Hello")
6      public:
7          int anInteger;
8          void sayHello();
9          void sayHello(char *);
10         #pragma SOMMethodName(sayHello(char *), "sayHello_string")
11     };

```

Generated IDL for Class Hello (file `hellow.idl`):

```

1  #ifndef __hellow_idl
2  #define __hellow_idl
3  /*
4   *
5   * Generated on Sat Mar 16 19:22:33 1996
6   * Generated from hellow.hh
7   * Using IBM VisualAge C++ for OS/2, Version 3
8   */
9  #include <somobj.idl>

```

Continued

```

10 interface Hello;
11 interface Hello : SOMObject {
12     void sayHello ();
13     void sayHello_string (in string p_arg1);
14 #ifdef __SOMIDL__
15     implementation {
16         align=4;
17         long anInteger;
18         anInteger: cxxmap="anInteger",offset=0,align=4,size=4,
19             nonstaticaccessors,public,cxxdecl="int anInteger;";
20         sayHello: public,nonstatic,cxxmap="sayHello()",
21             cxxdecl="void sayHello();";
22         sayHello_string: public,nonstatic,cxxmap="sayHello(char*)",
23             cxxdecl="void sayHello(char*)";
24         somDefaultConstVAssign: public,override;
25         somDefaultConstAssign: public,override;
26         somDefaultConstVCopyInit: public,override,init;
27         somDefaultInit: public,override,init;
28         somDestruct: public,override;
29         somDefaultCopyInit: public,override;
30         somDefaultConstCopyInit: public,override;
31         somDefaultVCopyInit: public,override;
32         somDefaultAssign: public,override;
33         somDefaultVAssign: public,override;
34         declarationorder = "anInteger, sayHello, sayHello_string,
35             somDefaultConstVAssign, somDefaultConstAssign,
36             somDefaultConstVCopyInit, somDefaultInit,
37             somDestruct";
38         releaseorder:
39             sayHello,
40             sayHello_string,
41             anInteger;
42         callstyle = idl;
43         dtsclass;
44         directinitclasses = "SOMObject";
45         cxxmap = "Hello";
46         cxxdecl = "class Hello : public virtual SOMObject";
47     };
48 #endif
49 };
50 #endif /* __helloworld_idl */

```

Private and Protected

The IDL generated for a DirectToSOM C++ is known as *usage bindings*. This means that only information required by a client to use the class is supplied. Private and protected class information is generated differently from public class information. In some cases, such as private base classes, there is no indication of that inheritance relationship in the generated IDL at all. For others cases, such as private member functions, their declarations are guarded by an `#ifdef __PRIVATE__` or `#ifdef __PROTECTED__` directive, which by default

will not be processed by the SOM IDL compiler. This prevents any unnecessary private class information from being made generally available.

In order to process the private information, the SOM compiler **-p** option can be used when compiling the IDL. However, this option is typically used only for generating the implementation bindings for classes described in IDL, and implemented using the language bindings. You should not need to use it for DirectToSOM C++. You may want to expose protected class information so that it is available for derived classes. For this situation, you should specify **-D__PROTECTED__** (which defines the macro) on the SOM compile command to process only protected information, and not private.

The following example shows how IDL is generated for class A, which contains private and protected members and a private base class. Note that, except in the **directInitClasses** list at line 85, the private base class **Base** does not appear anywhere in the generated IDL, not even in the **cxxdecl** for the class because the fact that **Base** is a base class for **A** is not necessary information in order for clients or derived classes to use **A**.

The interface declaration for the private member function **foo()** is guarded by **#ifdef __PRIVATE__**, at line 13, while the protected member function method **bar()** is guarded by **#if defined(__PROTECTED__) || defined(__PRIVATE__)** at line 15 in the **.idl** file. The qualified modifiers for each member in the implementation statement include the modifier **private**, **protected**, and **public** as appropriate. To allow the release order for the class to remain consistent, but to prevent making private and protected names visible, the release order names are also guarded at lines 58 through 81. If the appropriate guard macro is defined, the real name for the member is included in the release order. Otherwise, a dummy name is supplied using the convention **s__Pn**, which acts as a placeholder, but prevents the real name from being accessible.

DirectToSOM C++ Class Base (file base.hh):

```
1 #include <som.hh>
2
3 class Base : public SOMObject {
4     #pragma SOMClassName(*, "Base")
5 };
```

DirectToSOM C++ Class A (file classa.hh):

```
1 #include <som.hh>
2 #include "base.hh"
3
4 class A : private Base {
5     #pragma SOMNoMangling(*)
```

Continued

```

6      #pragma SOMClassName(*, "A")
7      int a1;
8      static int a2;
9      void foo();
10     protected:
11         int a3;
12         virtual void bar();
13     public:
14         int a4;
15         virtual void bar(int);
16         #pragma SOMMethodName(bar(int), "bar_int")
17     private:
18         int a5;
19 };

```

Generated IDL for Class A (file *classa.idl*):

```

1  #ifndef __classa_idl
2  #define __classa_idl
3  /*
4   *
5   * Generated on Sat Mar 23 16:12:26 1996
6   * Generated from classa.hh
7   * Using IBM VisualAge C++ for OS/2, Version 3
8   */
9  #include <somobj.idl>
10 #include <base.idl>
11 interface A;
12 interface A : SOMObject {
13     #ifdef __PRIVATE__
14         void foo ();
15     #endif
16     #if defined(__PROTECTED__) || defined(__PRIVATE__)
17         void bar ();
18     #endif
19     void bar_int (in long p_arg1);
20     #ifdef __SOMIDL__
21         implementation {
22             align=4;
23             long a1;
24             a1: cxxmap="a1",offset=8,align=4,size=4,
25                 nonstaticaccessors,private,cxxdecl="int a1;";
26             long a2;
27             a2: private,staticdata,cxxdecl="static int a2;";
28             foo: private,nonstatic,cxxmap="foo()",
29                 cxxdecl="void foo();";
30             long a3;
31             a3: cxxmap="a3",offset=4,align=4,size=4,
32                 nonstaticaccessors,protected,cxxdecl="int a3;";
33             bar: protected,cxxmap="bar()",
34                 cxxdecl="virtual void bar();";
35             long a4;
36             a4: cxxmap="a4",offset=0,align=4,size=4,
37                 nonstaticaccessors,public,cxxdecl="int a4;";

```



```

38     bar_int: public, cxxmap="bar(int)",
39     cxxdecl="virtual void bar(int);";
40     long a5;
41     a5: cxxmap="a5", offset=12, align=4, size=4,
42         nonstaticaccessors, private, cxxdecl="int a5;";
43     somDefaultConstVAssign: public, override;
44     somDefaultConstAssign: public, override;
45     somDefaultConstVCopyInit: public, override, init;
46     somDefaultInit: public, override, init;
47     somDestruct: public, override;
48     somDefaultCopyInit: public, override;
49     somDefaultConstCopyInit: public, override;
50     somDefaultVCopyInit: public, override;
51     somDefaultAssign: public, override;
52     somDefaultVAssign: public, override;
53     declarationorder = "a1, a2, foo, a3, bar, a4, bar_int, a5,
54         somDefaultConstVAssign, somDefaultConstAssign,
55         somDefaultConstVCopyInit, somDefaultInit,
56         somDestruct";
57     releaseorder:
58     #ifdef __PRIVATE__
59         a2,
60         foo,
61     #else
62         s__P0, s__P1,
63     #endif
64     #if defined(__PROTECTED__) || defined(__PRIVATE__)
65         bar,
66     #else
67         s__P2,
68     #endif
69         bar_int,
70         a4,
71     #if defined(__PROTECTED__) || defined(__PRIVATE__)
72         a3,
73     #else
74         s__P3,
75     #endif
76     #ifdef __PRIVATE__
77         a1,
78         a5
79     #else
80         s__P4, s__P5
81     #endif
82     ;
83     callstyle = idl;
84     dtsclass;
85     directinitclasses = "SOMObject, Base";
86     cxxmap = "A";
87     cxxdecl = "class A : public virtual SOMObject";
88     };
89 #endif
90 };
91 #endif /* __classa_idl */

```

Inheritance

In the previous example, the class `Base` was mostly invisible in the generated IDL because it is a private base class. If you have a public base class, more information will be generated in the IDL to indicate this relationship. The following programming example is a simple example. `Base` is a public base class of `Derived`. In the generated IDL, `Base` appears on the interface definition for the class at line 12 as a base class, and in the `cxxdecl` modifier at line 34 as a public base class. The `.idl` file containing the definition of `Base` is also included at line 10.

DirectToSOM C++ Class Base (file base.hh):

```
1 #include <som.hh>
2
3 class Base : public SOMObject {
4     #pragma SOMClassName(*, "Base")
5 };
```

DirectToSOM C++ Class Derived (file derived.hh):

```
1 #include <som.hh>
2 #include "base.hh"
3
4 class Derived : public Base {
5     #pragma SOMClassName(*, "Derived")
6 };
```

Generated IDL for Derived (file derived.idl):

```
1 #ifndef __derived_idl
2 #define __derived_idl
3 /*
4  *
5  * Generated on Fri Mar 22 08:53:55 1996
6  * Generated from derived.hh
7  * Using IBM VisualAge C++ for OS/2, Version 3
8  */
9 #include <somobj.idl>
10 #include <base.idl>
11 interface Derived;
12 interface Derived : Base {
13     #ifdef __SOMIDL__
14         implementation {
15             align=0;
16             somDefaultConstVAssign: public, override;
17             somDefaultConstAssign: public, override;
18             somDefaultConstVCopyInit: public, override, init;
19             somDefaultInit: public, override, init;
20             somDestruct: public, override;
21             somDefaultCopyInit: public, override;
```

```

22         somDefaultConstCopyInit: public, override;
23         somDefaultVCopyInit: public, override;
24         somDefaultAssign: public, override;
25         somDefaultVAssign: public, override;
26         declarationorder = "somDefaultConstVAssign,
27             somDefaultConstAssign, somDefaultConstVCopyInit,
28             somDefaultInit, somDestruct";
29         releaseorder:
30     ;
31         callstyle = idl;
32         dtsclass;
33         directinitclasses = "SOMObject, Base";
34         cxxmap = "Derived";
35         cxxdecl = "class Derived : public Base";
36     };
37 #endif
38 #endif /* __derived_idl */

```

Methods

Kinds of Methods

A C++ class may have three different kinds of methods: virtual, nonvirtual, and static. These correspond to SOM method types as follows:

C++	SOM
virtual	static
nonvirtual	nonstatic
static	direct-call procedure

It can be somewhat confusing that *static* in SOM means *virtual* in C++, but you typically don't have to deal with the SOM definitions anyway. Unless otherwise specified, I will use the C++ terminology when discussing these method types.

By default, SOM methods are always virtual, so a C++ virtual method does not require a special modifier to indicate its type in the generated IDL. A nonvirtual C++ method is indicated by the modifier **nonstatic** in the generated IDL, while a static C++ method is indicated by the **procedure** modifier. In addition, because static C++ methods are not passed an instance pointer or an **Environment** pointer, the modifiers **noself** and **noenv** are also generated.

When a C++ virtual function is overridden in a derived class, it is represented in the generated IDL using the **override** modifier, which we have already seen used for the overriding of the 10 special **SOMObject** methods. Nonvirtual C++ methods cannot be overridden (in C++ or in SOM), only hidden. If a derived class contains a method signature with the same name as a nonvirtual base class method, this is represented in the IDL using the **reintroduce** modifier.

The following example shows how the various kinds of methods are represented in the generated IDL. The classes `Base` and `Derived` both define the methods `virtualMethod`, `nonVirtualMethod`, and `staticMethod`. In the generated IDL for `Base`, `virtualMethod` is defined with just the **public**, **cxxdecl**, and **cxxmap** modifiers, indicating that it is a C++ virtual, or SOM static, function. `nonVirtualMethod` includes an additional modifier, **non-static**, indicating that it is a C++ nonvirtual, or SOM nonstatic, function. `staticMethod` has the modifiers **procedure**, **noself**, and **noenv**, indicating that it is a C++ static, or SOM direct-call procedure, method. Each method appears in the release order as they were declared in the class.

The generated IDL for `Derived` is similar, but includes information indicating the methods are either overridden or reintroduced. Since `virtualMethod` is a virtual function, repeating its signature in the derived class is an override of that method, and is indicated as such by the modifier **override** in the IDL. Both `nonVirtualMethod` and `staticMethod` are nonvirtual, so repeating their signatures in the derived class is a reintroduction of those methods, which is indicated using the **reintroduce** modifier. Only the latter two methods appear in the release order, because the release order includes only methods that are introduced, or reintroduced, in the current class. Since `virtualMethod` is a virtual function, it is considered introduced in, and will appear only in the release order of, the first base class that supplies its signature. In this example, that class is `Base`.

DirectToSOM C++ Class Base (file methodb.hh):

```

1  #include <som.hh>
2
3  #pragma SOMNoMangling(on)
4
5  class Base : public SOMObject {
6      #pragma SOMClassName(*, "Base")
7      public:
8          virtual virtualMethod();
9          nonVirtualMethod();
10         static staticMethod();
11 };

```

DirectToSOM C++ Class Derived (file methodb.hh):

```

1  #include <som.hh>
2  #include "methodb.hh"
3
4  class Derived : public Base {
5      #pragma SOMClassName(*, "Derived")
6      public:
7          virtual virtualMethod();
8          nonVirtualMethod();
9          static staticMethod();
10 };

```

Generated IDL for Class Base (file methodb.idl):

```

1  #ifndef __methodb_idl
2  #define __methodb_idl
3  /*
4  *
5  * Generated on Fri Mar 22 08:59:49 1996
6  * Generated from methodb.h
7  * Using IBM VisualAge C++ for OS/2, Version 3
8  */
9  #include <somobj.idl>
10 interface Base;
11 interface Base : SOMObject {
12     long virtualMethod ();
13     long nonVirtualMethod ();
14     long staticMethod ();
15 #ifdef __SOMIDL__
16     implementation {
17         align=0;
18         virtualMethod: public,cxxmap="virtualMethod()",
19             cxxdecl="virtual int virtualMethod()";
20         nonVirtualMethod: public,nonstatic,
21             cxxmap="nonVirtualMethod()",
22             cxxdecl="int nonVirtualMethod()";
23         staticMethod: public,procedure,noself,noenv,
24             cxxmap="staticMethod()",
25             cxxdecl="static int staticMethod()";
26         somDefaultConstVAssign: public,override;
27         somDefaultConstAssign: public,override;
28         somDefaultConstVCopyInit: public,override,init;
29         somDefaultInit: public,override,init;
30         somDestruct: public,override;
31         somDefaultCopyInit: public,override;
32         somDefaultConstCopyInit: public,override;
33         somDefaultVCopyInit: public,override;
34         somDefaultAssign: public,override;
35         somDefaultVAssign: public,override;
36         declarationorder = "virtualMethod, nonVirtualMethod,
37             staticMethod, somDefaultConstVAssign,
38             somDefaultConstAssign, somDefaultConstVCopyInit,
39             somDefaultInit, somDestruct";
40         releaseorder:
41             virtualMethod,
42             nonVirtualMethod,
43             staticMethod;
44         callstyle = idl;
45         dtsclass;
46         directinitclasses = "SOMObject";
47         cxxmap = "Base";
48         cxxdecl = "class Base : public virtual SOMObject";
49     };
50 #endif
51 };
52 #endif /* __methodb_idl */

```

Generated IDL for Class Derived (file method.idl):

```

1  #ifndef __method_idl
2  #define __method_idl
3  /*
4   *
5   * Generated on Fri Mar 22 08:58:57 1996
6   * Generated from method.hh
7   * Using IBM VisualAge C++ for OS/2, Version 3
8   */
9  #include <somobj.idl>
10 #include <methodb.idl>
11 interface Derived;
12 interface Derived : Base {
13 #ifdef __SOMIDL__
14     implementation {
15         align=0;
16         virtualMethod: public, override,
17             cxxdecl="virtual int virtualMethod();";
18         nonVirtualMethod: public, nonstatic,
19             cxxmap="nonVirtualMethod()", reintroduce,
20             cxxdecl="int nonVirtualMethod();";
21         staticMethod: public, procedure, noself, noenv,
22             cxxmap="staticMethod()", reintroduce,
23             cxxdecl="static int staticMethod();";
24         somDefaultConstVAssign: public, override;
25         somDefaultConstAssign: public, override;
26         somDefaultConstVCopyInit: public, override, init;
27         somDefaultInit: public, override, init;
28         somDestruct: public, override;
29         somDefaultCopyInit: public, override;
30         somDefaultConstCopyInit: public, override;
31         somDefaultVCopyInit: public, override;
32         somDefaultAssign: public, override;
33         somDefaultVAssign: public, override;
34         declarationorder = "virtualMethod, nonVirtualMethod,
35             staticMethod, somDefaultConstVAssign,
36             somDefaultConstAssign, somDefaultConstVCopyInit,
37             somDefaultInit, somDestruct";
38         releaseorder:
39             nonVirtualMethod,
40             staticMethod;
41         callstyle = idl;
42         dtsclass;
43         directinitclasses = "SOMObject, Base";
44         cxxmap = "Derived";
45         cxxdecl = "class Derived : public Base";
46     };
47 #endif
48 };
49 #endif /* __method_idl */

```

Operators

Operator methods are translated into IDL in the same way as standard methods, and can be used from other language as such. The main caveat is that the **SOMNoMangling** pragma does not apply to operators, so you must explicitly supply method names for each operator method if they are to be used in a cross-language environment. A good convention to follow for naming operator methods is to use the operator name followed by the types that it accepts. For example, the C++ method declarations:

```
int operator=(int);
#pragma SOMMethodName(operator=(int), "assignInteger")
int operator==(int);
#pragma SOMMethodName(operator==(int), "isEqualToInteger")
```

will result in the IDL method declarations:

```
long assignInteger (in long p_arg1);
long isEqualToInteger (in long p_arg1);
```

Parameter Directional Attributes (in/out/inout)

The CORBA IDL specification requires that method parameters be declared as **in**, **out**, or **inout**, indicating that the parameter is to be passed from client to server, from server to client, or in both directions, respectively. The DirectToSOM compiler generates only **in** or **inout** directional attributes for parameters. If the argument represents an address, the parameter is generated as **inout**, otherwise it is generated as **in**. Exceptions are `char *` and addresses of SOM object instances. A `char *` is mapped to an IDL **string** type (I will discuss this mapping in the next section), and is passed as **in**. SOM objects are implicitly passed by reference in IDL, so a parameter that is the address of a SOM object will be generated as **in** also. But pointers to `char *` or SOM objects will be passed as **inout**. For example, the method declarations for the following class:

```
class A : public SOMObject {
    foo(int, int *);
    foo2(char, char *, char **);
    bar(A, A*, A**);
};
```

would be generated in the IDL interface section as:

```
long foo (in long p_arg1, inout long p_arg2);
long foo2 (in char p_arg1, in string p_arg2, inout string p_arg3);
long bar (in A p_arg1, in A p_arg2, inout A p_arg3);
```

Inline Methods

The bodies for inline methods are not generated as part of the method declaration in the generated IDL. All inline methods are generated out-of-line, and do not form part of the class interface.

Mapping Types from C++ to IDL

Most C++ types can be mapped directly to IDL types, but there are some cases where such a mapping cannot take place because either IDL does not support a certain type or the IDL type is not exactly the same as the C++ type. In addition, there may be certain IDL types that you want to generate from DirectToSOM C++ for which there is no equivalent C++ type, such as IDL sequences.

Type names are not mangled as are member and class names. If you do have collisions in type names that differ only by case, you must use the **SOMIDLDecl** pragma to specify the desired IDL declaration for the given type. The following list shows the type mapping that occurs from C++ to IDL.

<i>C++ Type</i>	<i>IDL Type</i>
char	char
signed char	char (will likely change to SOMFOREIGN in next release)
unsigned char	octet (8-bit value guaranteed not to be converted)
short	short
int	long or short
long	long
signed	signed (for short, long)
unsigned	unsigned (for short, long)
float	float
double	double
void	void
long double	SOMFOREIGN
enum	integer typedef with constants
SOM class	interface
SOM class with friend or containing SOMFOREIGN members	SOMFOREIGN
native C++ structs/unions/class	SOMFOREIGN
C struct (no methods)	struct

<code>type_expr *</code>	<code>translated_type_expr *</code>
<code>type_expr &</code>	<code>translated_type_expr *</code>
<code>char *</code>	<code>string</code>
pointer to function or member	<code>SOMFOREIGN</code>
array	<code>array</code>

Most of the translations are fairly straightforward, but a few merit more detailed discussion.

IDL String Type

The IDL **string** type is an array of characters terminated by a NULL character. The distinction between an IDL array of characters and an IDL **string** is that an IDL array has a fixed upper bound, whereas an IDL **string** may not. The IDL **string** type is essentially the same as a C++ string literal. When generating IDL, the compiler translates `char *` in C++ to a **string** type in IDL. For example, given the following class:

```
class Hello : public SOMObject {
    char *aString;
    void sayHello(char *);
};
```

the members `aString` and `sayHello` would be generated in the IDL file as:

```
void sayHello (in string p_arg1);
...
string aString;
```

A typedef for a **string** type is defined when the SOM header files are included in a DirectToSOM C++ program, so you can use this type directly in your DirectToSOM C++ code; for example, `string aString`, or just use `char *`.

Enumerations

Although there is an IDL enum type defined in the CORBA standard, it is different from the C++ enum. An IDL enumeration begins at 1, whereas a C++ enumeration begins at 0 by default. Each value in an IDL enumeration is represented as a 4-byte quantity, while a C++ enumeration value occupies 1, 2, or 4 bytes as required to represent the full range of value. Finally, you cannot specify other values for an IDL enumeration, whereas in C++ you can assign arbitrary values to each enumerator.

The DirectToSOM C++ compiler translates an enum into a 1, 2, or 4-byte type, with associated constant declarations for the enumeration values. For example, the following enumerations:

```
enum E {a, b};
enum E2 {a2, b2=300};
```

are translated to IDL as shown:

```
typedef octet E;
const E a = 0;
const E b = 1;
#pragma modifier E: cxxdecl="enum E {a, b};",
    impctx = "C++", size = 1, align = 1;
#pragma modifier a: cxxdecl="";
#pragma modifier b: cxxdecl="";
typedef unsigned short E2;
const E2 a2 = 0;
const E2 b2 = 300;
#pragma modifier E2: cxxdecl="enum E2 {a2, b2 = 300};",
    impctx = "C++", size = 2, align = 2;
#pragma modifier a2: cxxdecl="";
#pragma modifier b2: cxxdecl="";
```

The first enumeration can be represented in a single byte, so it is mapped to an octet, whereas the second enumeration requires 2 bytes to hold the value 300, so it is represented as a short. Note that the **impctx** modifier is not actually required and will likely be removed in a future release.

SOMFOREIGN Types

Any C++ type that cannot be represented in IDL is declared as an IDL **SOMFOREIGN** type. **SOMFOREIGN** is a catchall type that is used to describe declarations that are outside the IDL type system. A C++ type that maps to a **SOMFOREIGN** type cannot typically be used from other languages, unless that language can represent that type accurately. **SOMFOREIGN** types may not be portable even between different DirectToSOM C++ compiler implementations, because the underlying storage model may not be consistent. Further, additional work is required to use **SOMFOREIGN** types with distributed DSOM (prior to SOMObjects 3.0, DSOM did not even support **SOMFOREIGN** types). So, if you are going to share a class with other languages or use it with DSOM, you should avoid **SOMFOREIGN** types whenever possible.

As an example of a **SOMFOREIGN** type, the following C++ function pointer type:

```
typedef void (*funcPtr) (int);
```

would be translated to IDL as:

```
typedef SOMFOREIGN sf0__;
#pragma modifier sf0__: impctx = "C++", size = 4, align = 4,
```

```

    cxxdecl="typedef void(* funcPtr)(int);";
#pragma somemittypes on
typedef sf0__ funcPtr;
#pragma modifier funcPtr: size = 4,align = 4,
    cxxdecl="typedef void(* funcPtr)(int);";
#pragma somemittypes off

```

The **importx** modifier is required for **SOMFOREIGN** types, and indicates the source language of the type. (The DirectToSOM compiler currently generates **SOMFOREIGN** types as two pieces, which causes problems when generating IDL for other languages. This is likely to be fixed in the next release of the products to generate just a single **SOMFOREIGN** type definition using the actual type name only.)

Nested Classes

Nested types in C++ are a purely syntactic mechanism in which the type name is scoped to the containing class, avoiding name collisions. A nested type definition does not add anything semantically to the containing class, and does not affect the size of instances of that class. Because IDL does not support nested interfaces, when a DirectToSOM class contains a nested SOM class definition, the nested class is generated at file scope in the IDL and assigned a C-scoped name to avoid collisions. The assigned name is the containing class, with two underscores, followed by the class name. This is the same model used by the emitter when generating language bindings for IDL classes that contain nested type definitions. In addition, because a nested class may have dependencies upon a type nested inside a class, all other nested types must be flattened to file scope also.

When a nested SOM class is flattened, the modifiers **nested** and **nefts** are generated in the contained and containing classes respectively, which indicates the original nesting relationship. In addition, for purposes of regenerating an **.hh** file, a **dts_fwd** is included in the class definition, which will map to a forward declaration in the regenerated **.hh** file for each nested class type. This is done in the same way as the forward declarations at file scope, to handle any references to that type before it is defined. (Only nested classes for which forward definitions may have appeared in the original class are included in the **dts_fwd**.)

The following example illustrates how nested SOM classes are handled. Note the **nested** modifier at line 34 of the IDL file for class **outer::inner**, and the **nefts** modifier at line 58 for class **outer**.

DirectToSOM C++ Class Outer (file nested.hh):

```

1 #include <som.hh>
2
3 class outer : public SOMObject {

```

Continued

```

4     public:
5         class inner: public SOMObject {
6         };
7     };

```

Generated IDL for Outer (file nested.idl):

```

1  #ifndef __nested_idl
2  #define __nested_idl
3  /*
4  *
5  * Generated on Sat Mar 23 19:24:31 1996
6  * Generated from nested.hh
7  * Using IBM VisualAge C++ for OS/2, Version 3
8  */
9  #include <somobj.idl>
10 interface outer;
11 interface outer;
12 interface outer__inner : SOMObject {
13 #ifdef __SOMIDL__
14     implementation {
15         align=0;
16         somDefaultConstVAssign: public,override;
17         somDefaultConstAssign: public,override;
18         somDefaultConstVCopyInit: public,override,init;
19         somDefaultInit: public,override,init;
20         somDestruct: public,override;
21         somDefaultCopyInit: public,override;
22         somDefaultConstCopyInit: public,override;
23         somDefaultVCopyInit: public,override;
24         somDefaultAssign: public,override;
25         somDefaultVAssign: public,override;
26         declarationorder = "somDefaultConstVAssign,
27             somDefaultConstAssign, somDefaultConstVCopyInit,
28             somDefaultInit, somDestruct";
29         releaseorder:
30     };
31     callstyle = idl;
32     dtsclass;
33     directinitclasses = "SOMObject";
34     nested = "outer";
35     cxxmap = "inner";
36     cxxdecl = "class inner : public virtual SOMObject";
37 };
38 #endif
39 };
40 interface outer : SOMObject {
41     typedef SOMFOREIGN dts__fwd__outer;
42 #ifdef __SOMIDL__
43     implementation {
44         align=0;
45         dts__fwd__outer: impctx="C++",
46             cxxdecl="public : class inner;\n";
47         outer__inner: public;

```

```

48     somDefaultConstVAssign: public,override;
49     somDefaultConstAssign: public,override;
50     somDefaultConstVCopyInit: public,override,init;
51     somDefaultInit: public,override,init;
52     somDestruct: public,override;
53     somDefaultCopyInit: public,override;
54     somDefaultConstCopyInit: public,override;
55     somDefaultVCopyInit: public,override;
56     somDefaultAssign: public,override;
57     somDefaultVAssign: public,override;
58     nests = "outer__inner";
59     declarationorder = "dts_fwd_outer, outer__inner,
60         somDefaultConstVAssign, somDefaultConstAssign,
61         somDefaultConstVCopyInit, somDefaultInit,
62         somDestruct";
63     releaseorder:
64 ;
65     callstyle = idl;
66     dtsclass;
67     directinitclasses = "SOMObject";
68     cxxmap = "outer";
69     cxxdecl = "class outer : public virtual SOMObject";
70 };
71 #endif
72 };
73 #endif /* __nested_idl */

```

Mapping by File

By default, the DirectToSOM compiler generates corresponding IDL declarations only for types that are defined in the current file. If a type appears in a separate file, it will not be generated in the IDL for the current file, the assumption being that the type will be included separately. For each **.hh** file included in the current file, a corresponding **.idl** file will be included in the generated **.idl** file, which will handle most type dependencies. But if the type appears in an **.h** file for example, a corresponding **#include** will not be generated in the **.idl** file. If that **.h** file contains a type upon which the current file depends, you may need to have that type definition generated directly into the **.idl** file, even if it was not defined in the current file. For such situations, you can use the **SOMIDLTypes** pragma.

In the following example, class **A** depends upon the type **myType**, which is defined in the file **idltype2.h**. Because **idltype.h** is not an **.hh** file, a corresponding include of an **.idl** file will not be generated in the **.idl** file for class **A**. In order to resolve the dependency upon **myType**, the **SOMIDLTypes** pragma is used to force the compiler to generate **myType** into the **.idl** file.

Type Definition for myType (file idltype2.h):

```
1 typedef int myType;
```

DirectToSOM C++ Class A (file idltypes.hh):

```

1  #include <som.hh>
2
3  #include "idltype2.h"
4
5  class A : public SOMObject {
6      #pragma SOMIDLTypes(*, myType)
7      myType m;
8  };

```

Generated IDL for Class A (file idltypes.idl):

```

1  #ifndef __idltypes_idl
2  #define __idltypes_idl
3  /*
4   *
5   * Generated on Mon May 6 07:45:52 1996
6   * Generated from idltypes.hh
7   * Using IBM VisualAge for C++ for Windows, Version 3.5
8   */
9  #include <somobj.idl>
10 interface za;
11 #pragma somemittypes on
12 typedef long myType;
13 #pragma modifier myType: cxxdecl="typedef int myType;";
14 #pragma somemittypes off
15 interface za : SOMObject {
16     #ifdef __SOMIDL__
17         implementation {
18             align=4;
19             myType m;
20             m: cxxmap="m",offset=0,align=4,size=4,
21                 nonstaticaccessors,private,cxxdecl="myType m;";
22             somDefaultConstVAssign: public,override;
23             somDefaultConstAssign: public,override;
24             somDefaultConstVCopyInit: public,override,init;
25             somDefaultInit: public,override,init;
26             somDestruct: public,override;
27             somDefaultCopyInit: public,override;
28             somDefaultConstCopyInit: public,override;
29             somDefaultVCopyInit: public,override;
30             somDefaultAssign: public,override;
31             somDefaultVAssign: public,override;
32             declarationorder = "m, somDefaultConstVAssign,
33                 somDefaultConstAssign, somDefaultConstVCopyInit,
34                 somDefaultInit, somDestruct";
35             releaseorder:
36             #ifdef __PRIVATE__
37                 m
38             #else
39                 s__P0
40             #endif

```

```

41 ;
42     callstyle = idl;
43     dtsclass;
44     directinitclasses = "SOMObject";
45     cxxmap = "A";
46     cxxdecl = "class A : public virtual SOMObject";
47 };
48 #endif
49 };
50 #endif /* __idltypes_idl */
51

```

Other Mappings

Attributes

CORBA attributes declared in DirectToSOM C++ using the **SOMAttribute** pragma are mapped to IDL attribute declarations in the interface section. The generated IDL attribute declaration consists of three parts: the declaration itself, a modifier for the attribute, and release order entries for the attribute methods and data (if the attribute has associated backing data). The modifiers **public**, **protected**, and **private** are used to indicate the access of the attribute backing data, while **publicaccessors**, **protectedaccessors**, and **privateaccessors** are used to indicate the access of the attributes **_get/_set** methods. By default, the attribute **_get/_set** methods are virtual in SOM, but nonvirtual in DirectToSOM C++. The modifier **nonstaticaccessors** is generated by the compiler to indicate when the methods are nonvirtual. The attribute **_get/_set** methods appear in the release order, followed by the backing data, if any.

The following example shows the IDL generated for various different types of attributes. The backing data for each attribute is given the same access as the corresponding attribute methods using the **publicdata** and **protecteddata** keywords in the **SOMAttribute** pragma. (The **SOM-ReleaseOrder** pragma is necessary to circumvent a problem in the generation of the release order for **nodata** attributes.)

The corresponding attribute declarations appear in the **.idl** file at lines 13 through 19. The **readonly** keyword is used at line 16 to indicate that **constAttrib** will have no **_set** method. The modifiers for each attribute appear at lines 23 through 33. Each includes modifiers to indicate the access of both the **_get/_set** methods and the backing data, which are public by default. The modifier lists for all except **virtualAttrib** include the **nonstaticaccessors** modifier, indicating that its corresponding **_get/_set** methods are virtual.

The **releaseorder** modifier appears at lines 48 through 75. The **_get/_set** methods appear first, from lines 49 through 64, followed by the backing data, from lines 65 through 75. There is no backing data entry for

the attribute `noData`. Each entry for the private and protected attributes is guarded by macros, so that the actual names will not be available unless the `.idl` file is compiled with private information visible.

DirectToSOM C++ Class A (file attrib.hh):

```

1  #include <som.hh>
2
3  class A : public SOMObject {
4      #pragma SOMNoMangling(*)
5      public:
6          int attrib;
7          #pragma SOMAttribute(attrib, publicdata)
8          int noDataAttrib;
9          #pragma SOMAttribute(noDataAttrib, nodata)
10     protected:
11         char *constAttrib;
12         #pragma SOMAttribute(constAttrib, readonly, protecteddata)
13     private:
14         int virtualAttrib;
15         #pragma SOMAttribute(virtualAttrib, virtualaccessors, \
16                               privatedata)
17
18         #pragma SOMReleaseOrder(attrib, noDataAttrib, constAttrib, \
19                                   virtualAttrib)
20 };

```

Generated IDL for Class A (file attrib.idl):

```

1  #ifndef __attrib_idl
2  #define __attrib_idl
3  /*
4   *
5   * Generated on Sat Mar 23 11:13:11 1996
6   * Generated from attrib.hh
7   * Using IBM VisualAge C++ for OS/2, Version 3
8   */
9  #include <somobj.idl>
10 interface za;
11 interface za : SOMObject {
12     #ifdef __PRIVATE__
13         attribute long virtualAttrib;
14     #endif
15     #if defined(__PROTECTED__) || defined(__PRIVATE__)
16         readonly attribute string constAttrib;
17     #endif
18         attribute long attrib;
19         attribute long noDataAttrib;
20     #ifdef __SOMIDL__
21         implementation {
22             align=4;
23             attrib: cxxmap="attrib",offset=0,align=4,size=4,

```



```

24         nonstaticaccessors,public,publicaccessors,
25         cxxdecl="int attrib;";
26     noDataAttrib: cxxmap="noDataAttrib",nodata,
27         cxxdecl="int noDataAttrib;";
28     constAttrib: cxxmap="constAttrib",offset=4,align=4,size=4,
29         nonstaticaccessors,protected,protectedaccessors,
30         cxxdecl="char* constAttrib;";
31     virtualAttrib: cxxmap="virtualAttrib",offset=8,align=4,
32         size=4,private,privateaccessors,
33         cxxdecl="int virtualAttrib;";
34     somDefaultConstVAssign: public,override;
35     somDefaultConstAssign: public,override;
36     somDefaultConstVCopyInit: public,override,init;
37     somDefaultInit: public,override,init;
38     somDestruct: public,override;
39     somDefaultCopyInit: public,override;
40     somDefaultConstCopyInit: public,override;
41     somDefaultVCopyInit: public,override;
42     somDefaultAssign: public,override;
43     somDefaultVAssign: public,override;
44     declarationorder = "attrib, noDataAttrib, constAttrib,
45         virtualAttrib, somDefaultConstVAssign,
46         somDefaultConstAssign, somDefaultConstVCopyInit,
47         somDefaultInit, somDestruct";
48     releaseorder:
49         _get_attrib,
50         _set_attrib,
51         _get_noDataAttrib,
52         _set_noDataAttrib,
53     #if defined(__PROTECTED__) || defined(__PRIVATE__)
54         _get_constAttrib,
55     #else
56         s__P1,
57     #endif
58         s__P0,
59     #ifdef __PRIVATE__
60         _get_virtualAttrib,
61         _set_virtualAttrib,
62     #else
63         s__P2, s__P3,
64     #endif
65         attrib,
66     #if defined(__PROTECTED__) || defined(__PRIVATE__)
67         constAttrib,
68     #else
69         s__P4,
70     #endif
71     #ifdef __PRIVATE__
72         virtualAttrib
73     #else
74         s__P5
75     #endif
76     ;
77     callstyle = idl;
78     dtsclass;

```

Continued

```

79         directinitclasses = "SOMObject";
80         cxxmap = "A";
81         cxxdecl = "class A : public virtual SOMObject";
82     };
83 #endif
84 };
85 #endif /* __attrib_idl */

```

Embedded Objects

In DirectToSOM C++, you can declare a class data member to be an instance of a SOM object. This is known as an *embedded* SOM object. As discussed in Chapter 5, when an instance of a SOM object is declared as a data member in a class, storage is allocated for that embedded member at the end of the containing class. In IDL, when an instance variable is declared to be of a SOM class type, that variable is always treated as a reference to that type. IDL has no concept of the instance actually being embedded in the class itself as in DirectToSOM C++.

In order to represent an embedded member in IDL, the DirectToSOM compiler maps embedded object declarations to a combination of an object reference plus a **SOMFOREIGN** type. The object reference can be used by other languages to access the embedded member, and **SOMFOREIGN** provides information to the IDL compiler about the embedded type. An **embedded** modifier is generated for the object reference to indicate that it represents an embedded object, while **SOMFOREIGN** contains an **impctx** modifier indicating that it represents an instance of the embedded type. Because the underlying type is a SOM class, this is one of the few cases where the use of a **SOMFOREIGN** type is portable between different DirectToSOM C++ compilers.

As an example, the following programming example shows the IDL generated for the embedded members `a` and `arrOfA` in the class `Container`. At line 13 in the generated IDL, the interface contains a type definition of the **SOMFOREIGN** type `embo__0`, which is used to represent the embedded object `a`, while `embo__1` at line 14 is used for `arrOfA`. The qualified modifier for `a`, at line 19, contains the modifier `embedded=embo__0`, indicating that `embo__0` is the corresponding embedded object instance for this object reference. `embo__0` is declared at line 21 as an instance of the **SOMFOREIGN** type `embo__0`. Then, at line 22, the **impctx** modifier for the **SOMFOREIGN** type `embo__0` is `"SOM, instance(A)"` indicating that **SOMFOREIGN** represents an embedded instance of class `A`. A similar set of declarations is generated for the embedded array `arrOfA`, using the **SOMFOREIGN** `embo__1` instead.

DirectToSOM C++ Class A (file embeda.hh):

```

1  #include <som.hh>
2
3  class A : public SOMObject {

```

```

4   #pragma SOMClassName(*, "A")
5   #pragma SOMNoMangling(*)
6   int i;
7   };

```

DirectToSOM C++ Class Container (file embed.hh):

```

1  #include <som.hh>
2
3  #include "embeda.hh"
4
5  class Container : public SOMObject {
6      #pragma SOMClassName(*, "Container")
7      #pragma SOMNoMangling(*)
8  public:
9      A a;
10     A arrOfA[10];
11 };

```

Generated IDL for Class Container (file embed.idl):

```

1  #ifndef __embed_idl
2  #define __embed_idl
3  /*
4   *
5   * Generated on Thu Mar 21 19:54:16 1996
6   * Generated from embed.hh
7   * Using IBM VisualAge C++ for OS/2, Version 3
8   */
9  #include <somobj.idl>
10 #include <embeda.idl>
11 interface Container;
12 interface Container : SOMObject {
13     typedef SOMFOREIGN embo__0;
14     typedef SOMFOREIGN embo__1;
15 #ifdef __SOMIDL__
16     implementation {
17         align=4;
18         A a;
19         a: cxxmap="a",offset=0,align=4,size=4, nonstaticaccessors,
20           public,embedded=emb__0,cxxdecl="A a;";
21         embo__0 emb__0;
22         embo__0: impctx = "SOM, instance(A)";
23         A arrOfA;
24         arrOfA: cxxmap="arrOfA",offset=4,align=4,size=40,
25               nonstaticaccessors,public,embedded=emb__1,
26               cxxdecl="A arrOfA[10];";
27         embo__1 emb__1[10];
28         embo__1: impctx = "SOM, instance(A)";
29         somDefaultConstVAssign: public,override;
30         somDefaultConstAssign: public,override;
31         somDefaultConstVCopyInit: public,override,init;
32         somDefaultInit: public,override,init;

```

Continued

```

33     somDestruct: public, override;
34     somDefaultCopyInit: public, override;
35     somDefaultConstCopyInit: public, override;
36     somDefaultVCopyInit: public, override;
37     somDefaultAssign: public, override;
38     somDefaultVAssign: public, override;
39     declarationorder = "a, arrOfA, somDefaultConstVAssign,
40         somDefaultConstAssign, somDefaultConstVCopyInit,
41         somDefaultInit, somDestruct";
42     releaseorder:
43         a,
44         arrOfA;
45     callstyle = idl;
46     dtsclass;
47     directinitclasses = "SOMObject";
48     cxxmap = "Container";
49     cxxdecl = "class Container : public virtual SOMObject";
50 };
51 #endif
52 };
53 #endif /* __embed_idl */

```

Constructors

C++ constructors are mapped to IDL either as overrides of one of the special **SOMObject** initializer methods (see Chapter 6), or as newly introduced IDL *initializer* methods. An IDL initializer method is indicated with the modifier **init**, and always accepts an initializer controller as the first parameter. (Originally, this controller was defined in IDL as **inout somInitCtrl**, but was subsequently changed to **in som3InitCtrl** because **inout** parameters cannot be passed as null, which is a requirement when calling initializer methods explicitly. This will be discussed in more detail in Chapter 9, *Interlanguage Object Sharing*. However, some DirectToSOM compilers, including the version I wrote this book with, may still be generating **inout somInitCtrl**.)

Constructors that don't map to one of the **SOMObject** initializers should be explicitly given a SOM name for interlanguage use, as the default compiler-generated name is not very readable. As an example, the programming following shows class A with three constructors. The default constructor and the copy constructor at lines 6 and 7 will map to **SOMObject** methods **somDefaultInit** and **somDefaultConstCopyInit**, shown at lines 16 and 17 in the generated IDL file. The third constructor at line 8 will be generated as a newly introduced initializer method, with the SOM name **init_int** specified with the **SOMMethodName** pragma at line 9. **init_int** is introduced in the interface section of the IDL at line 12, with a qualified modifier at lines 19 and 20.

DirectToSOM C++ Class A (file ctr.hh):

```

1  #include <som.hh>
2
3  class A : public SOMObject {
4      #pragma SOMClassName("A")
5      public:
6          A();
7          A(const A&);
8          A(int);
9          #pragma SOMMethodName(A(int), "init_int")
10 };

```

Generated IDL for Class A (file ctr.idl):

```

1  #ifndef __ctr_idl
2  #define __ctr_idl
3  /*
4   *
5   * Generated on Thu Mar 21 20:24:26 1996
6   * Generated from ctr.hh
7   * Using IBM VisualAge C++ for OS/2, Version 3
8   */
9  #include <somobj.idl>
10 interface A;
11 interface A : SOMObject {
12     void init_int (inout somInitCtrl ctrl, in long p_arg1);
13 #ifdef __SOMIDL__
14     implementation {
15         align=0;
16         somDefaultInit: public,override,init,cxxdecl="A()";
17         somDefaultConstCopyInit: public,override,init,
18             cxxdecl="A(const A&)";
19         init_int: public,nonstatic,init,cxxmap="A(int)",
20             cxxdecl="A(int)";
21         somDefaultConstVAssign: public,override;
22         somDefaultConstAssign: public,override;
23         somDestruct: public,override;
24         somDefaultCopyInit: public,override;
25         somDefaultAssign: public,override;
26         somDefaultVAssign: public,override;
27         declarationorder = "somDefaultInit, somDefaultConstCopyInit,
28             init_int, somDefaultConstVAssign,
29             somDefaultConstAssign, somDestruct";
30         releaseorder:
31             init_int;
32         callstyle = idl;
33         dtsclass;
34         directinitclasses = "SOMObject";
35         cxxmap = "A";
36         cxxdecl = "class A : public virtual SOMObject";
37     };

```

Continued

```

38 #endif
39 };
40 #endif /* __ctr_idl */

```

Templates

As discussed in Chapter 5, *Programming Considerations*, each instantiation of a DirectToSOM template class corresponds to one SOM class definition. This is reflected in the IDL generated. For the most part, templates are generated just like any other SOM class. For example, the following shows the template class `Stack`, which is instantiated on line 16 using the **define** pragma. Each template instantiation must be assigned a unique name, if a compiler-generated default is not desirable, as shown at line 17.

SOMFOREIGN at lines 10 through 16 is used to cause the template forward declaration "template <class T> class Stack" to be regenerated into the `.hh` file. This will likely be replaced by a **passthru** statement in a future release. Note that the template controlling class is not generated in the IDL, just each instantiation.

DirectToSOM C++ Class Stack (file template.hh):

```

1 #include <som.hh>
2
3 template <class T> class Stack : public SOMObject {
4     #pragma SOMNoMangling(*)
5     T *stack;
6     int sz;
7     public:
8     Stack(int s);
9     #pragma SOMMethodName(Stack(int), "InitInt")
10    ~Stack();
11    void push(T elem);
12    T pop();
13    int size();
14 };
15
16 #pragma define(Stack<int>)
17 #pragma SOMClassName(Stack<int>, "IntegerStack")

```

Generated IDL for Class Stack (file template.idl):

```

1 #ifndef __template_idl
2 #define __template_idl
3 /*
4  *
5  * Generated on Thu Mar 21 21:03:24 1996
6  * Generated from template.hh
7  * Using IBM VisualAge C++ for OS/2, Version 3
8  */

```

```

9  #include <somobj.idl>
10  typedef SOMFOREIGN sf0__;
11  #pragma modifier sf0__ : impctx = "C++", size = 0, length = 0,
12      align = 0, cxxdecl="template <class T> class Stack;";
13  #pragma somemittypes on
14  typedef sf0__ sf0__template;
15  #pragma modifier sf0__template : size = 0, length = 0, align = 0,
16      cxxdecl="template <class T> class Stack;";
17  #pragma somemittypes off
18  interface IntegerStack;
19  interface IntegerStack : SOMObject {
20      void InitInt (inout somInitCtrl ctrl, in long p_arg1);
21      void push (in long p_arg1);
22      long pop ();
23      long size ();
24  #ifdef __SOMIDL__
25      implementation {
26          align=4;
27          long * stack;
28          stack: cxxmap="stack",offset=0,align=4,size=4,
29              nonstaticaccessors,private,cxxdecl="int* stack;";
30          long sz;
31          sz: cxxmap="sz",offset=4,align=4,size=4,
32              nonstaticaccessors,private,cxxdecl="int sz;";
33          InitInt: public,nonstatic,init,cxxmap="Stack(int)",
34              cxxdecl="Stack(int);";
35          somDestruct: public,override,cxxdecl="virtual ~Stack();";
36          push: public,nonstatic,cxxmap="push(int)",
37              cxxdecl="void push(int);";
38          pop: public,nonstatic,cxxmap="pop()",
39              cxxdecl="int pop();";
40          size: public,nonstatic,cxxmap="size()",
41              cxxdecl="int size();";
42          somDefaultConstVAssign: public,override;
43          somDefaultConstAssign: public,override;
44          somDefaultConstVCopyInit: public,override,init;
45          somDefaultCopyInit: public,override;
46          somDefaultConstCopyInit: public,override;
47          somDefaultVCopyInit: public,override;
48          somDefaultAssign: public,override;
49          somDefaultVAssign: public,override;
50          declarationorder = "stack, sz, InitInt, somDestruct,
51              push, pop, size, somDefaultConstVAssign,
52              somDefaultConstAssign, somDefaultConstVCopyInit";
53          releaseorder:
54              InitInt,
55              push,
56              pop,
57              size,
58  #ifdef __PRIVATE__
59              stack,
60              sz
61  #else
62              s__P0, s__P1

```

Continued

```

63 #endif
64 ;
65         callstyle = idl;
66         dtsclass;
67         directinitclasses = "SOMObject";
68         cxxmap = "Stack<int>";
69         cxxdecl = "class Stack<int> : public virtual SOMObject";
70         somDefaultInit: nocall;
71     };
72 #endif
73 };
74 #endif /* __template__idl */

```

Generating IDL-Specific Information from C++

General Modifiers

There are situations where you will want to generate information into an IDL file that does not map from any specific DirectToSOM C++ construct. This is typically necessary when using one of the SOM frameworks, such as DSOM. For such situations, you can use the **SOMIDLPass** pragma, which allows you to generate arbitrary strings into the IDL file. These strings can be modifiers, type definitions, or any other valid IDL construct. You must generate the string with respect to a SOM class, and it can be done at a variety of different points within that class declaration: the beginning or end of the file, the beginning or end of the interface section for that class, or the beginning or end of the implementation section for the class. You can specify multiple **SOMIDLPass** pragmas for a class, and they will be cumulative.

A common use of this pragma is to specify the **dllname** modifier, which supplies the name of the DLL that will contain the current class. This information is required when the class is to be dynamically loaded. In the next example, **SOMIDLPass** is specified at line 5 in the C++ class definition, resulting in the **dllname** modifier appearing at line 38 in the generated IDL.

DirectToSOM C++ Class Hello (file idldecl.hh):

```

1 #include <som.hh>
2
3 class Hello : public SOMObject {
4     #pragma SOMClassName(*, "Hello")
5     #pragma SOMIDLPass(*, "Implementation-End", "dllname = \"hello.dll\";")
6     public:
7         void sayHello();
8 };

```


Generated IDL for Class Hello (file idldecl.idl):

```

1  #ifndef __idldecl_idl
2  #define __idldecl_idl
3  /*
4  *
5  * Generated on Fri Mar 22 20:00:13 1996
6  * Generated from idldecl.hh
7  * Using IBM VisualAge C++ for OS/2, Version 3
8  */
9  #include <somobj.idl>
10 interface Hello;
11 interface Hello : SOMObject {
12     void sayzhello__fv ();
13 #ifdef __SOMIDL__
14     implementation {
15         align=0;
16         sayzhello__fv: public,nonstatic,cxxmap="sayHello()",
17             cxxdecl="void sayHello()";
18         somDefaultConstVAssign: public,override;
19         somDefaultConstAssign: public,override;
20         somDefaultConstVCopyInit: public,override,init;
21         somDefaultInit: public,override,init;
22         somDestruct: public,override;
23         somDefaultCopyInit: public,override;
24         somDefaultConstCopyInit: public,override;
25         somDefaultVCopyInit: public,override;
26         somDefaultAssign: public,override;
27         somDefaultVAssign: public,override;
28         declarationorder = "sayzhello__fv, somDefaultConstVAssign,
29             somDefaultConstAssign, somDefaultConstVCopyInit,
30             somDefaultInit, somDestruct";
31         releaseorder:
32             sayzhello__fv;
33         callstyle = idl;
34         dtsclass;
35         directinitclasses = "SOMObject";
36         cxxmap = "Hello";
37         cxxdecl = "class Hello : public virtual SOMObject";
38         dllname = "hello.dll";
39     };
40 #endif
41 };
42 #endif /* __idldecl_idl */

```

Modifying Generated IDL Declarations

There are situations where the compiler-generated declarations aren't exactly what you would like. In such situations, you can use the **SOMIDLDecl** pragma to specify exactly the IDL declaration that should be generated for a given type, data, or function member.

A common use of this pragma is to generate CORBA directional attributes for method parameters. As discussed earlier, the DirectToSOM compiler generates only **in** or **inout** directional attributes for parameters. There is no way to indicate in the C++ source itself that a parameter should be **out**. To do this, you must use the **SOMIDLDecl** pragma to specify the desired signature. For example, given the following:

```
class A : public SOMObject {
    foo(int *);
        #pragma SOMMethodName(foo(int *), "foo")
        #pragma SOMIDLDecl(foo(int *, \
            "long foo {out long p_arg1}")
    foo2(char **);
        #pragma SOMMethodName(foo2(char **), "foo2")
        #pragma SOMIDLDecl(foo2(char **, \
            "long foo2 {out string p_arg1}")
};
```

the method declarations for `foo` and `foo2` will be generated in the **.idl** file as:

```
long foo {out long p_arg1};
long foo2 {out string p_arg1};
```

Note that you must also use either the **SOMMethodName** or the **SOMNoMangling** pragma to change the name of the method. The **SOMIDLDecl** only affects what is generated in the interface section for the declaration; it does not affect the SOM name, which will be used for qualified modifiers and so on.

Appending Modifiers

In addition to the **SOMIDLDecl** pragma, you can modify method declarations using the **SOMMethodAppend**. This is used to generate CORBA IDL **context** and **raises** expressions. Anything that you specify in the string for the pragma is appended to the end of the IDL method declaration. For example, the following declarations:

```
void foo(int);

#pragma SOMMethodAppend(foo(int), " context {\\"mycontext\\"}")
```

result in the IDL method declaration:

```
void foo (in long p_arg1) context("mycontext");
```

Sequences

IDL supports a sequence type, which is essentially an unbounded array. IDL arrays are bounded, unlike C++ arrays. An IDL sequence is defined as follows in IDL:

sequence <simple-type [*positive-integer-constant*];

where *simple-type* is any valid IDL type, and *positive-integer-constant* is an optional maximum bound for the sequence. For example, the following defines a sequence, `integer10`, of at most 10 integers:

```
typedef sequence <long, 10> integer10;
```

A sequence is generated into an **.hh** file as a structure containing three members: the maximum sequence length, the current sequence length, and a pointer to the sequence data. For example, the preceding sequence would be generated in an **.hh** file as:

```
#ifndef _IDL_SEQUENCE_long_defined
#define _IDL_SEQUENCE_long_defined
typedef struct {
    unsigned long _maximum;
    unsigned long _length;
    long *_buffer;
} _IDL_SEQUENCE_long;
#endif // _IDL_SEQUENCE_long_defined

typedef _IDL_SEQUENCE_long integer10;
```

But what if you want to define a sequence in your DirectToSOM C++ class and generate it into the published IDL file as a sequence? You can do this with the **SOMIDLPass** pragma. The easiest way is to define the sequence in IDL first, generate the **.hh** file, and copy the relevant parts into your class. Then, add a **SOMIDLPass** pragma, specifying the C++ type name and the sequence as you want it to be generated.

The following example shows how to do this. The IDL file `seqidl.idl` declares the sequence `integer10` as it is to be generated into the IDL from our DirectToSOM C++ source. This **.idl** file is used to generate an **.hh** file from which the C++ definition of the sequence is copied into the DirectToSOM C++ implementation file `seqhh.hh`. I have cleaned up the definition to define the structure with type name `integer10`, rather than having the extra `typedef`. The **SOMIDLPass** pragma at line 11 causes the given statement to be generated in the **.idl** file at the beginning of the interface, while **SOMIDLDecl** is used at line 13 to generate a dummy declaration for `integer10` to prevent any name conflicts with the declaration produced by the **SOMIDLPass** pragma. (Hopefully, a future version of the compiler will allow you to simply specify a null string for the **SOMIDLDecl** declaration, indicating that no declaration should be generated. Currently, specifying a null string causes the compiler to generate just a semicolon, which the IDL compiler flags as an error.) The resulting **.idl** file is partially shown in `seqhh.idl`.

You cannot simply use a **SOMIDLDecl** for `integer10` to specify the structure declaration. Because nested types are generated at file scope in the **.idl** file, using **SOMIDLDecl** would result in the type being generated at file

scope, rather than embedded in the interface, as shown in this programming example.

IDL Definition for Sequence Integer10 (file seqidl.idl):

```
1 #include <somobj.idl>
2
3 interface A : SOMObject {
4     typedef sequence <long, 10> integer10;
5 };
```

DirectToSOM C++ Code for Sequence Integer10 (file seqhh.hh):

```
1 #include <som.hh>
2
3 class A : public SOMObject {
4     #pragma SOMClassName(*, "A")
5     public:
6         struct integer10 {
7             unsigned long _maximum;
8             unsigned long _length;
9             long *_buffer;
10        };
11     #pragma SOMIDLPass(*, "Interface-Begin", \
12         "typedef sequence <long, 10> integer10;")
13     #pragma SOMIDLDecl(integer10, "typedef char Dummy")
14 };
```

Generated IDL for Sequence Integer10 (file seqhh.idl):

```
1 #ifndef __seqhh_idl
2 #define __seqhh_idl
3 /*
4  *
5  * Generated on Mon Mar 18 18:29:01 1996
6  * Generated from seqhh.hh
7  * Using IBM VisualAge C++ for OS/2, Version 3
8  */
9 #include <somobj.idl>
10 interface A;
11 typedef char Dummy;
12 interface A : SOMObject {
13     typedef sequence <long, 10> integer10;
14     .
15     .
16     .
```

Generating DirectToSOM Class Definitions from IDL Definitions

There are two distinct situations where you may want to generate an **.hh** file from an **.idl** file. The first is to produce DirectToSOM C++ client bindings

for a class that is not implemented in DirectToSOM C++, so that the class can be used from DirectToSOM C++. The second is to port an existing class from IDL to DirectToSOM C++.

Generating DirectToSOM C++ Client Bindings

You can generate a DirectToSOM C++ client binding file from an **.idl** file through the SOM **hh** emitter. This is invoked using the SOM compiler **sc** command, specifying **-shh** to indicate that a DirectToSOM C++ class definition should be generated. When using **-shh**, you must always specify the **-mnoqualifytypes** option also. This indicates that the SOM compiler should not produce C-scoped names (where the scope name is explicitly prepended to the original name) in the data structures that it passes to the **hh** emitter, because name scoping is achieved through nesting in DirectToSOM C++. In addition, you should ensure that the **SMINCLUDE** environment variable is defined to include the SOM header files, and your own. For example,

```
set SMINCLUDE=.;%SOMBASE%\include
```

will set the SOM compiler include search path to include the current directory and the include directory below the SOM installation directory (assuming that **SOMBASE** is set to the SOM installation directory).

As an example, the following shows the resulting DirectToSOM C++ client binding file generated from the IDL definition of the class **hello**. This definition can be used by DirectToSOM C++ clients of a class implemented using another language. Much of the file is straightforward, but there are a few items of interest. At line 19, the **SOMNonDTS** pragma is turned on for the duration of the class, and turned off at line 50. This pragma indicates that the class is not implemented by DirectToSOM C++. At lines 23 through 27, the two **SOMClassName** pragma specifications are guarded by **#if defined(__SOM__MODULES)**. (Both names specified for **SOMClassName** are the same because the original class did not appear in an IDL module; this will become more clear in the next section, in which I discuss the mapping from IDL modules.) Line 28 specified no mangling for the class, which is sufficient for an IDL-defined class, because there cannot be any overloaded names. Lines 29 through 33 are redundant.

At lines 36 through 43, definitions for a default constructor, copy constructor, destructor, and assignment method are given for the class. These correspond to the methods that the SOM emitter generates by default for the C and C++ language bindings. Even though the original IDL did not specify these methods, they are still available for the class. One of the two methods **somAssign** or **somDefaultAssign** will be processed depending upon the definition of the **__EXTENDED_SOM_ASSIGNMENTS__** macro. As discussed in Chapter 6, *Inside DirectToSOM C++*, not all

DirectToSOM C++ implementations support the **somAssign** form of overloading the SOM assignment methods, so the macro would be undefined for such products, and **somDefaultAssign** used instead.

IDL Definition for Class Hello (file helloidl.idl):

```

1  #include <somobj.idl>
2
3  interface Hello : SOMObject
4  {
5      void sayHello();
6      implementation {
7          releaseorder: sayHello;
8      };
9  };

```

Generated DirectToSOM C++ Client Binding for Hello (file helloidl.hh):

```

1  #ifndef _DTS_HH_INCLUDED_helloidl
2  #define _DTS_HH_INCLUDED_helloidl
3
4  /* Start Interface Hello */
5
6  // This file was generated by the IBM "DirectToSOM" emitter for C++
7  // Generated at 03/17/96 04:32:40 EST
8  // The efw file is version 1.62
9
10 #include <som.hh>
11
12 #pragma SOMAsDefault(on)
13 class SOMClass;
14 #pragma SOMAsDefault(pop)
15 #pragma SOMAsDefault(on)
16 class SOMObject;
17 #pragma SOMAsDefault(pop)
18 #include <somobj.hh>
19 #pragma SOMNonDTS(on)
20
21 class Hello : public ::SOMObject {
22
23 #if defined(__SOM__MODULES__)
24     #pragma SOMClassName(*, "Hello")
25 #else
26     #pragma SOMClassName(*, "Hello")
27 #endif
28     #pragma SOMNoMangling(*)
29     #pragma SOMNonDTS(*)
30
31     #pragma SOMCallstyle(idl)
32     #pragma SOMAsDefault(off)
33     #pragma SOMAsDefault(pop)
34

```

```

35     public :
36         Hello();
37         Hello(Hello&);
38         virtual ~Hello();
39 #ifdef __EXTENDED__SOM__ASSIGNMENTS__
40         virtual Hello& somAssign(Hello&);
41 #else
42         virtual SOMObject* somDefaultAssign(som3AssignCtrl*, SOMObject*);
43 #endif
44
45         virtual void sayHello();
46         #pragma SOMReleaseOrder { \
47                                     "sayHello"
48     };
49
50         #pragma SOMNonDTS(pop)
51 /* End Hello */
52 #endif /* _DTS_HH_INCLUDED_hello1 */

```

Modules

You can have multiple interface definitions in the same **.idl** file by enclosing the definitions inside a **module** statement. The SOM names of any classes inside a module become the module name prepended to the class name. The **hh** emitter will generate a **SOMModule** pragma in the resulting **.hh** file to indicate the module name, in which case the compiler must also use the combined module and class name in making references to the SOM class data structures. However, not all DirectToSOM C++ compilers support the **SOMModule** pragma yet, so the emitter generates two versions of the **SOMClassName** pragma in the generated **.hh** file, one for compilers that support **SOMModule**, and one for those that don't, guarded by **#if defined(__SOM__MODULES)**. For those that do support this pragma, the SOM class name is just the actual class name as given in the IDL file, as the compiler will prepend the module name; otherwise, the SOM class name is the module name prepended to the SOM class name.

The following programming example illustrates this, with the two classes **Hello** and **Hello2** inside the module **MyModule**. The example shows the syntax currently generated for the **SOMModule** pragma by the **hh** emitter; this is likely to change slightly in the next release of the products.

IDL Definition for Module *MyModule* (file *module.idl*):

```

1  #include <somobj.idl>
2
3  module MyModule {
4
5      interface Hello : SOMObject
6      {
7          void sayHello();
8          implementation {

```

Continued

```

9         releaseorder: sayHello;
10    };
11 };
12
13 interface Hello2 : SOMObject
14 {
15     void sayHello();
16     implementation {
17         releaseorder: sayHello;
18     };
19 };
20
21 };

```

***Generated DirectToSOM C++ Client Binding for MyModule
(file mymodule.hh):***

```

1  #ifndef _DTS_HH_INCLUDED_module
2  #define _DTS_HH_INCLUDED_module
3  // This file was generated by the IBM "DirectToSOM" emitter for C++
4  // Generated at 03/17/96 05:33:22 EST
5  // The efw file is version 1.62
6
7
8  /* Start Module MyModule */
9
10 #include <somobj.hh>
11 #include <somobj.hh>
12
13 class MyModule {
14 #if defined(__SOM__MODULES__)
15 #pragma SOMModule (MyModule)
16 #endif
17
18
19     public :
20     #pragma SOMNonDTS(on)
21
22     class Hello : public ::SOMObject {
23
24 #if defined(__SOM__MODULES__)
25     #pragma SOMClassName(*, "Hello")
26 #else
27     #pragma SOMClassName(*, "MyModule_Hello")
28 #endif
29     #pragma SOMNoMangling(*)
30     #pragma SOMNonDTS(*)
31
32     #pragma SOMCallstyle(idl)
33     #pragma SOMAsDefault(off)
34     #pragma SOMAsDefault(pop)
35

```



```

36     public :
37         Hello();
38         Hello(Hello&);
39         virtual ~Hello();
40 #ifdef __EXTENDED__SOM__ASSIGNMENTS__
41         virtual Hello& somAssign(Hello&);
42 #else
43         virtual SOMObject* somDefaultAssign(som3AssignCtrl*, SOMObject*);
44 #endif
45
46         virtual void sayHello();
47         #pragma SOMReleaseOrder ( \
48                                     "sayHello*"
49     );
50
51         #pragma SOMNonDTS(pop)
52         #pragma SOMNonDTS(on)
53
54     class Hello2 : public ::SOMObject {
55
56     #if defined(__SOM__MODULES__)
57         #pragma SOMClassName(*, "Hello2")
58     #else
59         #pragma SOMClassName(*, "MyModule_Hello2")
60     #endif
61         #pragma SOMNoMangling(*)
62         #pragma SOMNonDTS(*)
63
64         #pragma SOMCallstyle {idl}
65         #pragma SOMAsDefault(off)
66         #pragma SOMAsDefault(pop)
67
68     public :
69         Hello2();
70         Hello2(Hello2&);
71         virtual ~Hello2();
72 #ifdef __EXTENDED__SOM__ASSIGNMENTS__
73         virtual Hello2& somAssign(Hello2&);
74 #else
75         virtual SOMObject* somDefaultAssign(som3AssignCtrl*, SOMObject*);
76 #endif
77
78         virtual void sayHello();
79         #pragma SOMReleaseOrder ( \
80                                     "sayHello*"
81     );
82
83         #pragma SOMNonDTS(pop)
84     };
85
86     /* End MyModule */
87
88 #endif /* __DTS_HH_INCLUDED_module */

```

Porting an Existing IDL Class Definition to DirectToSOM C++

There are many subtleties involved in the mapping from DirectToSOM C++ to SOM. Given that you have an IDL definition for a class and you want to implement that class using DirectToSOM C++ but still make the IDL definition available, you currently need to port that class definition to DirectToSOM C++ and regenerate the IDL. It is expected that a future release of the SOM compiler will support modifiers to allow a DirectToSOM C++ class implementation header file to be generated and used directly from an IDL description. Until then, you should not attempt to use handwritten IDL to publish a class interface to a class that is implemented in DirectToSOM C++, because it is extremely difficult to ensure that the IDL definition accurately reflects the mapping that DirectToSOM C++ is performing. And if the IDL definition does not match exactly, you will most certainly encounter problems.

Probably the simplest way to port an existing IDL file to DirectToSOM C++ is to add the **dtsclass** modifier to the class and generate the **.hh** file. Specifying **dtsclass** indicates to the emitter that the class is implemented in DirectToSOM C++, and not to generate the class information that reflects the default SOM class definition. Then you can update the **.hh** file by hand to clean it up and apply the techniques described earlier to generate an IDL file that can be used from other languages.

For instance, the next programming example shows the class `Hello` for which we generated an **.hh** client binding file earlier, but this time the **dtsclass** modifier has been added to the implementation statement. The resulting **.hh** file contains none of the SOM-supplied method information that the client binding file did. The DirectToSOM C++ compiler may implicitly supply these methods, but they are not specified in the class definition unless the programmer explicitly supplies them. In addition, the generated **.hh** file does not contain the **SOMNonDTS** pragma, indicating that this class is implemented in DirectToSOM C++.

IDL Definition for Class `Hello` (file `helloid2.idl`):

```

1  #include <somobj.idl>
2
3  interface Hello : SOMObject
4  {
5      void sayHello();
6      implementation {
7          releaseorder: sayHello;
8          dtsclass;
9      };
10 };

```

Generated DirectToSOM C++ Client Binding for Hello (file helloid2.hh):

```

1  #ifndef _DTS_HH_INCLUDED_helloid2
2  #define _DTS_HH_INCLUDED_helloid2
3
4  /* Start Interface Hello */
5
6  // This file was generated by the IBM "DirectToSOM" emitter for C++
7  // Generated at 03/17/96 05:11:19 EST
8  // The efw file is version 1.62
9
10 #include <som.hh>
11
12     #pragma SOMAsDefault(on)
13 class SOMClass;
14     #pragma SOMAsDefault(pop)
15     #pragma SOMAsDefault(on)
16 class SOMObject;
17     #pragma SOMAsDefault(pop)
18 #include <somobj.hh>
19
20 class Hello : public ::SOMObject {
21
22     #if defined(__SOM_MODULES__)
23         #pragma SOMClassName(*, "Hello")
24     #else
25         #pragma SOMClassName(*, "Hello")
26     #endif
27     #pragma SOMNoMangling(*)
28     #pragma SOMCallstyle(idl)
29     #pragma SOMAsDefault(off)
30     #pragma SOMAsDefault(pop)
31
32     public :
33         virtual void sayHello();
34         #pragma SOMReleaseOrder ( \
35                                     "sayHello")
36 };
37
38 /* End Hello */
39 #endif /* _DTS_HH_INCLUDED_helloid2 */

```

The Interface Repository

The SOM compiler can optionally create a database, called the SOM Interface Repository (IR), which contains class information as supplied by the IDL description. The database can be queried through the SOM API so that a program, at run time, can access any information available about a class interface. The interface repository content and programming interface

conforms to that defined by OMG's CORBA Interface Repository. Among other things, the IR provides another mechanism for programming languages to support interaction with SOM, along with information needed by the DSOM framework.

IDL declarations are compiled into the interface repository using the SOM compiler **sc** command, specifying the **-sir** and **-u** options. In addition to the **SINCLUDE** environment variable described earlier, the SOM compiler will update the rightmost interface repository file specified with the **SOMIR** environment variable. You should always specify your own interface repository file as the rightmost file for update purposes (and to make the SOM-supplied interface repository read-only). For example:

```
set SOMIR=%SOMIR%;\mydir\mysom.ir;
```

Then, to generate an IDL file into the interface repository, run the command:

```
sc -sir -u hello.idl
```

To see what is generated into the IR for a given class, type the **command IRDUMP classname**.

In subsequent chapters covering, for example, interlanguage object sharing and distributed SOM, I will cover the use of the interface repository in more detail.

Distributed SOM

DSOM Overview

Distributed SOM (DSOM) allows you to write applications that share objects between processes or even between machines. For the most part, the fact that an object is in a different address space is transparent to both the class client and its implementation. The distribution is handled implicitly by the DSOM framework. DSOM operates in one of two modes, Workstation DSOM and Workgroup DSOM, which support distribution of objects between processes on the same machine or across a network respectively. This chapter describes the DSOM support available with SOMObjects 3.0 (DSOM 3.0), but highlights the differences between DSOM 3.0 and earlier versions (DSOM 2.x). (Note: The examples were tested using the SOMObjects 3.0 beta, which was only available for OS/2 at the time of writing, so the examples have been tested only on OS/2.)

The underpinning of the DSOM model is that of an object *proxy*, which is a local “shadow” object for a remote “real” object (see Figure 8.1). When you create a distributed DSOM object, the real object is created and maintained by a remote server process, and the local client process is given a proxy object to operate on. From the client perspective, the proxy object is manipulated in much the same way as a normal local object would be. The

DSOM run time transforms operations on the proxy object into operations on the remote object, and handles parameter data copying between the two. As shown in Figure 8.1, the client program can manipulate both local and remote objects, even of the same class.

Simple DSOM Example

Before getting into the details of DSOM, let's look at a simple example to illustrate how DSOM works. The following example shows the definition and implementation of the `Hello` class that we have seen previously. This time however, the test program, `tsthello.cpp`, creates and manipulates both a local and a remote `Hello` object.

At line 3 in the `tsthello.cpp` file, the DSOM DirectToSOM header file `<somdd.hh>` is included, containing the definitions for the DSOM classes, methods, and functions. Lines 11 and 12 create a local `Hello` object and invoke the method `sayHello` against that object, which will write "Hello world" to standard output. This is all pretty normal so far.

Line 15 is where the program starts to get interesting. Here the DSOM environment is initialized by calling the function `SOMD_Init`, passing the `__SOMEnv` variable as a parameter. At line 19, the `somdCreate` function is called to create a remote object and its proxy. Three parameters can be

DSOM Application

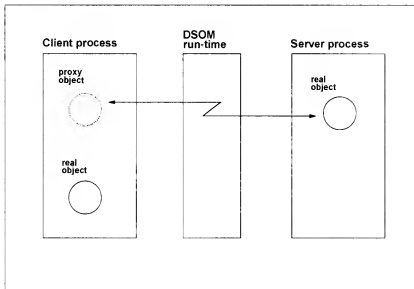


FIGURE 8.1 DSOM overview.

passed to **somdCreate**, the first being the **Environment** structure, the second the class name for the remote object, in this case **Hello**, and the third a **Boolean**, indicating whether the object should be initialized (that is, created with **somNew** or **somNewNoInit**). If the **somdCreate** method executes successfully, an instance of the **Hello** class will be created in a server process, and a proxy object for this remote object will be assigned to the variable **remoteObj**. This proxy object can be treated pretty much as if it were a local object; the client program does not need to know where the object exists in order to manipulate it. The **checkError** function, from file **check.h**, is invoked at line 20 to ensure that the object was successfully created.

Line 23 then invokes the method **sayHello** against the proxy object. DSOM will turn this into a method invocation against the remote object, which will result in "Hello world" being written to standard output at the server process. Line 26 deallocates both objects, using the **SOMObject** method **somFree**. (In DSOM 2.x, **somFree** only deallocates the remote object. In DSOM 3.0, it deallocates both the remote and the proxy objects.)

The program then uninitializes the DSOM environment, deletes the **__SOMEnv** storage, and terminates. And, from a coding perspective, that's all there is to it! There is some environment setup required, however, which I will discuss in the next section, to explain the commands at the end of the **makefile**. Also note that there is little error handling being done by the program, other than checking that the remote object was created successfully. You should always check the result of each DSOM operation, but more on that later.

Definition of DirectToSOM C++ Class Hello (hello\hello.hh):

```

1  #include <som.hh>
2
3  class Hello : public SOMObject {
4      #pragma SOMClassName(*, "Hello")
5      #pragma SOMIDLPass(*, "Implementation-End", \
6          "dllname = \"hello.dll\";")
7  public:
8      Hello() { }
9      void sayHello();
10 };

```

Implementation of DirectToSOM C++ Class Hello (hello\hello.cpp):

```

1  #include <fstream.h>
2  #include "hello.hh"
3
4  void Hello::sayHello()
5  {
6      cout << "Hello world" << endl;
7  }

```

Client of DirectToSOM C++ Class Hello (hello\stshello.cpp):

```

1  #include <iostream.h>
2  #include <assert.h>
3  #include <somd.hh>
4  #include "check.h"
5  #include "hello.hh"
6
7  int main(int argc, char *argv[])
8  {
9      __SOMEnv = SOM_CreateLocalEnvironment();
10
11     Hello localObj;
12     localObj.sayHello();
13
14     // Initialize DSOM run-time environment
15     SOMD_Init(__SOMEnv);
16
17     // create remote object
18     Hello *remoteObj =
19         (Hello *)somdCreate(__SOMEnv, "Hello", TRUE);
20     assert(! checkError(__SOMEnv, TRUE));
21
22     // Use proxy to issue message
23     remoteObj->sayHello();
24
25     // delete proxy and remote object
26     remoteObj->somFree();
27
28     SOMD_Uninit(__SOMEnv);
29     SOM_DestroyLocalEnvironment(__SOMEnv);
30
31     return(0);
32 }

```

Supporting Functions (include\check.h):

```

1  #include <iostream.h>
2  #include <somd.hh>
3
4  static boolean checkError(Environment *env,
5                          boolean dump = FALSE)
6  {
7      if (env->_major == NO_EXCEPTION)
8          return FALSE;
9      if (dump)
10         cout << "exception: " << somExceptionId(env)
11         << endl;
12     somExceptionFree(env);
13     return TRUE;
14 }

```


Makefile (hello\makefile):

```

1  all: hello.dll tsthello.exe somdimpl.dat
2
3  ICCOPTS = /I ..\include
4
5  hello.dll: hello.cpp hello.hh hello.i.c
6      icc $(ICCOPTS) /Ti+ /Ge- /B"/NOE" \
7          hello.cpp hello.i.c hello.def
8      implib hello.lib hello.dll
9
10 tsthello.exe: tsthello.cpp
11     icc $(ICCOPTS) /Ti+ tsthello.cpp hello.lib
12
13 hello.idl: hello.hh
14     icc $(ICCOPTS) hello.hh
15     sc -sir -u hello.idl
16
17 hello.i.c: hello.idl
18     sc -simod hello.idl
19     sc -sdef hello.idl
20
21 somdimpl.dat:
22     regimpl -D -i HelloServer
23     regimpl -A -i HelloServer
24     regimpl -a -i HelloServer -c Hello

```

DSOM Environment Setup

For DirectToSOM C++, the DSOM run time needs two types of information about a class in order to create a remote object. This information is gathered from two repositories: the interface repository and the *implementation repository*. The interface repository, as discussed in Chapter 7, is a database that contains complete information about a class interface, including the DLL name for the class implementation. DSOM uses the information in the interface repository to determine the method signatures—that is, the parameter types and return values—in order to manage the exchange of data between a proxy and a remote object.

For classes built using C or C++ bindings generated by the SOM 3.0 emitter, the interface repository is not used to retrieve detailed class information because the bindings generate *marshaling information* within the class binary that allows DSOM to access class information without it. This support is not yet available for DirectToSOM C++, so the interface repository is still used by DSOM for retrieving class information. It is expected that the next release of the compiler products will generate marshaling information. In any case, the interface repository is still used by DSOM for certain situations, even if marshaling information is generated, such as loading the class DLL.

DSOM Configuration

If this is the first time you are creating a DSOM program on your system (required for SOMObjects Version 3.0 and higher), you will first need to configure the Naming Service support, which is used by DSOM. This is done through the **som_cfg** command. You can use either the SOM default directory for storing the DSOM configuration or specify your own. If you plan to use the SOM default directory, you can skip to step 2.

- Step 1. *Specifying your own DSOM configuration directory:* In order to define your own directory for the DSOM configuration, you need to update the **somenv.ini** file with some configuration information. Copy the file **%SOMBASE%\etc\somenv.ini** to your own directory and set the **SOMENV** environment variable to reference this file. For example, in OS/2 or Windows, type:

```
SET SOMENV=f:\jennifer\dsom\dsom30\somenv.ini;%SOMENV%
```

Next decide on a directory that you will use for holding all DSOM files, and edit the **somenv.ini** file to **SOMDDIR** to this location (search for **SOMDDIR** until you find one on a line starting with a 'semicolon'. Remove the 'semicolon' also). With the SOMObject 3.0 beta, the directory specified must be empty, but this may change in the official release of the product. For example:

```
SOMDDIR=f:\jennifer\dsom\dsom30\files
```

I also updated the **HOSTNAME** and **USERNAME** parameters to specify my machine and user ID.

- Step 2. *Now run the **som_cfg** command as follows:*

```
som_cfg -i
```

This will read the information in the **somenv.ini** file and set up the Naming Service configuration in the directory given by **SOMDDIR**. Two settings will be added to the **SOMENV** file: **SOMNMOBJREF** and **HOSTKIND** (the default **SOMENV** file is **%SOMBASE%\etc\somenv.ini**). If you ever need to change the configuration information, such as the **SOMDDIR** directory, you will need to remove these two settings from the **SOMENV** file and rerun **som_cfg** to reconfigure the Naming Service.

Interface Repository Setup

Assuming that the **SOMIR** path was set up as described in Chapter 7, the following command creates the interface repository information for the previous example (see line 14 and 15 in the makefile):

```
icc hello.hh

sc -sir -u hello.idl
```

Implementation Repository Setup

The *implementation repository* contains information about the server process for a given class. Every server implementation must be registered with DSOM before it can be used. Registration involves defining a server implementation alias name and associating information with that alias name. Each server implementation definition will be created as a separate process by the DSOM run time, which uses the implementation repository to determine how to create that process.

The implementation repository can be updated through a programming interface or through the **regimpl** or **pregimpl** utilities. To register the information for the class in the previous example in the implementation repository, I used the following commands:

```
regimpl -A -i HelloServer

regimpl -a -i HelloServer -c Hello
```

The first line defines a new server implementation alias named `HelloServer`. This name is not significant—it can be any name you choose. The DSOM run time will create a process of this name when it receives a request for the server implementation run time. The next line associates the class `Hello` with the server alias `HelloServer`. When DSOM receives a request to create an object of type `Hello`, it will look up and find the `HelloServer` registered in the implementation repository and use the information associated with this server implementation to create the server process. This operation may seem a little useless, as it doesn't appear to register much, but there is default information, such as the server program and class, associated with this server that DSOM will use and needs to have defined. In a later example, I will show how to associate additional information with a server alias.

The commands just given appear at lines 23 and 24 in the makefile for the previous example. In addition, line 22 of the makefile deletes any existing definitions for the server `HelloServer`, prior to adding the new definition. The `regimpl.dat` file name used to control the implementation repository update at line 21 can be any name. If the file exists, the implementation repository is not updated, otherwise, it is. Creating a dummy file with this name will prevent the implementation repository from being updated every time you build the program. The name itself is actually an artifact from DSOM 2.x, in which a file of this name was created when the implementation repository was updated.

Starting the DSOM Daemon

All you need to do to run this example is to start the DSOM daemon, **somdd**, if it is not already running, as follows:

```
start somdd
```

Then you should be able to run the **tsthellob** program. (Ensure that **hello.dll** can be found by both the client and the server process).

The previous is a Workstation application, in which the client and server process exist on the same machine. See the SOMObjects Programmer's Guide for information on setup for Workgroup applications.

The Server Process

When a program requests that an object be created, DSOM will either use an existing server process if an appropriate one is active, or create a new process for a given server implementation. If a new server process is to be created, then DSOM will start a process by running the server program specified in the implementation repository. Among other things, the server program is responsible for accepting and responding to client requests; creating, finding and destroying local objects; and the transformation between client requests and local object operations. The process of transforming client requests into method invocations on local objects is known as the *demarshaling* of those requests. The opposite action of transforming the results into responses to the client, is known as the *marshaling* of those results.

By default, the server program used for each server implementation process is the DSOM-supplied program **somdsvr**. When started, it will first notify the DSOM daemon that it is ready to receive requests, then it will enter into a loop, accepting and processing client requests. For example, Figure 8.2 shows the process and object layout for the program shown in the previous example. The client process is running the program **tsthellob**, and the remote process is running the program **somdsvr**. There is one local **Hello** object, **localObj**; the **remoteObj** object is a proxy for the remote **Hello** object that exists in the server process. You can write your own server program, which will be discussed in more detail later in this chapter.

Creating Remote Objects with a Factory

In the previous example, the program requested only that an object of class **Hello** be created; no mention was made of *where* that object should be created. A remote object is created through a *factory*, which is an object that is used to create and return object references. Typically, a factory is simply a class object, but factories are not restricted to just being class objects.

Hello Application

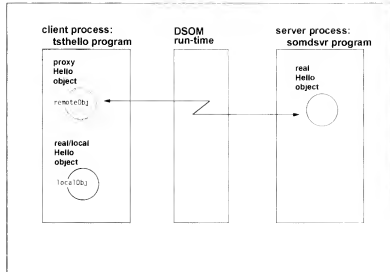


FIGURE 8.2 DSOM server process.

You can request that a remote object be managed by any factory that supports the class, as shown in the previous example, or you can request that an object be created through a specific factory. To do the latter, you must first obtain a proxy for a factory in the server process, and then request that the object be created through that factory.

Factories are located either directly through the Naming Service or through a Naming Service extension called the *factory service*. In this chapter, I will consider just the latter case. (Chapter 10, *The SOMObjects Object Services*, will discuss the use of the Naming Service to find factory objects.) First, you must obtain an object reference for the factory service, and through this object obtain a proxy for the factory object itself. The factory object is located through the **find_any** method of the factory service class **ExtendedNaming::ExtendedNamingContext**. This method is passed a string indicating constraints to be used in locating and returning the factory proxy. The factory service interprets the constraint string to determine the appropriate factory object to return.

For example, the following shows how to retrieve and use a factory to create a remote object. At line 17, the method **resolve_initial_references** is invoked against the object **SOMD_ORBObject**, with the parameter “**FactoryService**”. This will return a factory service object, stored in the variable **enc**, that represents the naming context where references to SOM object factories are stored. **SOMD_ORBObject** is an instance of the DSOM Object Request Broker (ORB) class. It provides a variety of methods for

both client and server support and is created during DSOM run-time environment initialization through **SOMD_Init**.

Next, the method **find_any** is invoked against `enc` at line 22, specifying a constraint string of "class == 'Hello' and alias == 'HelloServer'". This will cause the factory service to return a factory for the class `Hello` that is defined with a server alias of `HelloServer`. The server alias is the name used when the server implementation was registered, so this will pick up the server that we defined earlier with **regimpl**.

The factory is used to create a remote object at line 27 by invoking the method **somNew**. This will create an object in the server process and return a proxy to it. This object can be used exactly as the remote object in the previous example. In fact, the remote object in that first example was implicitly created through a factory. **somdCreate** implicitly calls **find_any**, specifying the given class as a constraint string. If you want to use a specific constructor instead of the default, you can invoke **somNewNoInit** against the factory object instead of **somNew**, followed by the specific constructor. (You must first resolve to the constructor method using the **SOMObject** method **somResolveByName** as described in Chapter 5, *Programming Considerations*.)

Once the remote object has been created, the factory object is no longer needed unless you want to create additional remote objects. At the end of the program, you should destroy the proxies for the factory service and the factory itself with the **SOMDObject** method **release**, as shown at lines 38 and 39, which does not destroy the corresponding remote objects. You should use **release**, rather than simply **delete** or **SOMFree**, because **release** will deallocate any communications resources that have been allocated for the proxy object. (While the **release** method appears to be valid only for **SOMDObject** objects, it is valid for any object. It is actually introduced in **SOMObject** in the private IDL, but it is public in **SOMDObject**. You can cast any object to **SOMDObject** and invoke **release**. For local objects, **release** has no effect, and for proxy objects, it deallocates only the proxy object, not the remote object.) Figure 8.3 shows the object and process layout for the program this example.

Creating an Object through a Factory (factory\sthello.cpp):

```

1  #include <iostream.h>
2  #include <assert.h>
3  #include <somd.hh>
4  #include "check.h"
5  #include "hello.hh"
6
7  int main(int argc, char *argv[])
8  {
9      __SOMEnv = SOM_CreateLocalEnvironment();
10
11     SOMD_Init(__SOMEnv);
12
```

Hello Application

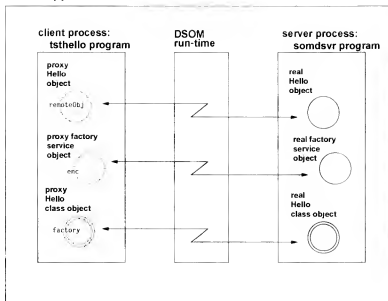


FIGURE 8.3 Process layout with factory.

```

13 // Get Naming Service factory service
14 ExtendedNaming::ExtendedNamingContext *enc =
15   (ExtendedNaming::ExtendedNamingContext *)
16   SOMD_ORBObject->
17     resolve_initial_reference("FactoryService");
18 assert(! checkError(__SOMEnv, TRUE));
19
20 // Find factory (class object) for class Hello
21 SOMClass *factory = (SOMClass *)
22   enc->find_any("class == 'Hello'"
23     " and alias == 'HelloServer'", 0);
24 assert(! checkError(__SOMEnv, TRUE));
25
26 // Create a remote object through the factory
27 Hello *remoteObj = (Hello *)factory->somNew();
28 assert(! checkError(__SOMEnv, TRUE));
29
30 // Use proxy to issue message
31 remoteObj->sayHello();
32
33 // Delete the proxy and remote object
34 remoteObj->somFree();
35
36 // Delete the factory/factory service proxies,
37 // but not the remote objects
38 ((SOMDObject *) factory)->release();

```

Continued

```

39      ((SOMDObject *)enc)->release();
40
41      SOMD_Uninit(__SOMEnv);
42      SOM_DestroyLocalEnvironment(__SOMEnv);
43
44      return(0);
45  }

```

Finding Existing Objects

The examples so far have always created a new object each time the program is run. But for many applications you may want to find and use an existing object. DSOM supports the ability to return a string representation for a given proxy, which you can then store in a file, for example. Later, that string can be read from the file, turned back into a proxy, and used as a remote object. This also allows different processes to share the same object. An alternative approach to saving and restoring a proxy string through DSOM is to use the Naming Service, which will be discussed in Chapter 10.

The following program illustrates the DSOM approach. At line 17, a number is read from standard input. At lines 19 and 20, a local object is created and the method `sayHello` is invoked, passing the input number. This will display the number on the screen. Rather than simply creating a remote object, at line 26, the function `ReadProxyFromFile` is called. If the function returns non-NULL, then an existing object proxy was found locally. Before this proxy can be used however, it must be checked to ensure the associated remote object is still valid. The remote object could be no longer in existence if the remote server process were terminated or the remote object were deleted through another application.

The `isValidRemoteObject` function has been added to `check.h` to invoke a method on the remote object and check the result to ensure that remote object is still valid. It returns true or false, indicating the validity of the proxy.

If the proxy is invalid or is not found locally, a remote object is created at line 36. This object is saved to a file using the function `WriteProxyToFile`. Once the proxy has either been found locally or created, the method `sayHello` is invoked against it, passing the input number. If you run the program from different processes, each will read a number from standard input and send the input number in the method invocation against the shared remote object.

The function `WriteProxyToFile`, at line 52, saves the ID for proxy to a file so that it can be accessed by other processes. At line 57, the **ORB** class method `object_to_string` is invoked against **SOMD_ORBObject**, passing the proxy object. It will convert the proxy into a string suitable for storing in a file. The string is then written to the given file, and control returns to the caller.

`ReadProxyFromFile`, at line 68, reads a saved stringified proxy and converts it back to a proxy object. The stringified proxy is read from the file at line 74. The **ORB** class method `string_to_object` is invoked at line 77,

passing the stringified proxy, which returns the corresponding proxy object. This proxy object is then returned to the caller.

Client of DirectToSOM C++ Class Hello (objid\tstobjid.cpp):

```

1  #include <iostream.h>
2  #include <fstream.h>
3  #include <assert.h>
4  #include <somd.hh>
5  #include "objid.hh"
6  #include "check.h"
7
8  void WriteProxyToFile(char *fname, SOMObject *remoteObj);
9  SOMObject* ReadProxyFromFile(char *fname);
10
11 int main(int argc, char *argv[])
12 {
13     __SOMEnv = SOM_CreateLocalEnvironment();
14
15     cout << "Enter a number: ";
16     long number;
17     cin >> number;
18
19     Objid localObj;
20     localObj.sayHello(number);
21
22     SOMD_Init(__SOMEnv);
23
24     Objid *remoteObj = NULL;
25     char *fname = "Objid.pxy";
26     remoteObj = (Objid *)ReadProxyFromFile(fname);
27
28     if (remoteObj && isValidRemoteObject(remoteObj))
29         cout << "Object found locally" << endl;
30     else {
31         // remote object not returned, so create
32         // object, and save the id for
33         // the object in the proxy file
34         cout << "Creating object" << endl;
35         remoteObj =
36             (Objid *)somdCreate(__SOMEnv, "Objid", TRUE);
37         assert(remoteObj && ! checkError(__SOMEnv, TRUE));
38         WriteProxyToFile(fname, remoteObj);
39     }
40
41     remoteObj->sayHello(number);
42
43     remoteObj->somFree();
44
45     SOMD_Uninit(__SOMEnv);
46     SOM_DestroyLocalEnvironment(__SOMEnv);
47
48     return(0);
49 }
```

Continued

```

50
51 // Writes the id for a proxy to the given file
52 void WriteProxyToFile(char *fname, SOMObject *remoteObj)
53 {
54     ofstream proxyFile(fname);
55     char *id;
56
57     id = SOMD_ORBObject->object_to_string(
58         (SOMDObject *)remoteObj);
59     proxyFile << id;
60     // free storage allocated by DSOM for return string
61     delete id;
62     proxyFile.close();
63 }
64
65
66 // Reads the id from the given file and
67 // returns corresponding proxy
68 SOMObject* ReadProxyFromFile(char *fname)
69 {
70     ifstream proxyFile(fname);
71     if (! proxyFile)
72         return NULL;
73     char proxyId[256];
74     proxyFile >> proxyId;
75     proxyFile.close();
76     SOMObject *remoteObj = (SOMObject *)
77         SOMD_ORBObject->string_to_object(proxyId);
78     return remoteObj;
79 }

```

Supporting Functions (include\check.h):

```

1  #include <iostream.h>
2  #include <somd.hh>
3
4  static boolean checkError(Environment *env,
5                          boolean dump = FALSE)
6  {
7      if (env->_major == NO_EXCEPTION)
8          return FALSE;
9      if (dump)
10         cout << "exception: " << somExceptionId(env)
11         << endl;
12     somExceptionFree(env);
13     return TRUE;
14 }
15
16 static boolean isValidRemoteObject(SOMObject *remoteObj)
17 {
18     if (remoteObj == NULL)
19         return FALSE;
20     char *tmp = remoteObj->somGetClassName();
21     if (checkError(somGetGlobalEnvironment())) {

```

```

22         cout << ">> Detected invalid object reference"
23         << endl << endl;
24         return FALSE;
25     }
26     // free returned memory
27     if (tmp)
28         delete tmp;
29     return TRUE;
30 }

```

DSOM Programming Considerations

The two major programming considerations when using DSOM with DirectToSOM C++ are data access and memory management. It is important to understand these concepts, otherwise you can introduce subtle bugs, and subsequently spend hours debugging problems.

Accessing Data Members

With DSOM, the actual object and its storage is created at the server side, and the client simply has a proxy object with which to interact with the remote object. This implies that the client cannot directly access data in the actual object, as the instance data is in a different address space, or perhaps even on another machine. You can still have public and protected data members in a DirectToSOM C++ class, but you must make them into CORBA attributes using the **SOMAttribute** pragma. Then the compiler will not attempt to directly access the data, but will instead use the **_get** and **_set** data access methods to access the data. (See Chapter 4 for a detailed discussion of the **SOMAttribute** pragma.)

In the next programming example show, at line 9 in *attrib.hh*, the **SOMAttribute** pragma is specified, which makes the data member *i* an attribute. The compiler will implicitly generate **_get_i** and **_set_i** methods that will return and update the value of *i* respectively. For any references to *i* outside the class, the compiler will call these methods, rather than access the data directly. So at lines 22 through 24 in *tstatr.cpp*, the references to the data member *i* will result in the attribute methods **_get_i** and **_set_i** being called, rather than the data being accessed directly.

When you make a data member into an attribute, by default the backing data (that is, the actual instance data) will become private, and the compiler will call the data access methods for any reference to the data member in a context that does not have access to private data for the class. References to attributes in the class implementation, which have access to private class data, will result in direct data access. For example, at line 6 in *attrib.cpp*, the data for member *i* will be accessed directly.

Definition of DirectToSOM C++ Class *Attrib* (*attrib\attrib.hh*):

```

1  #include <som.hh>
2
3  class Attrib : public SOMObject {
4      #pragma SOMClassName(*, "Attrib")
5      #pragma SOMIDLPass(*, "Implementation-End", \
6          "dllname = \"attrib.dll\";")
7  public:
8      int i;
9      #pragma SOMAttribute(i)
10     Attrib();
11 };

```

Implementation of DirectToSOM C++ Class *Attrib* (*attrib\attrib.cpp*):

```

1  #include <fstream.h>
2  #include "attrib.hh"
3
4  Attrib::Attrib()
5  {
6      i = 1;
7  }

```

Client of DirectToSOM C++ Class *Hello* (*attrib\stattr.cpp*):

```

1  #include <iostream.h>
2  #include <assert.h>
3  #include <somd.hh>
4  #include "check.h"
5  #include "attrib.hh"
6
7  int main(int argc, char *argv[])
8  {
9      __SOMEnv = SOM_CreateLocalEnvironment();
10
11     Attrib localObj;
12     cout << "Local object i is " << localObj.i << endl;
13     localObj.i = 5;
14     cout << "Local object i is " << localObj.i << endl;
15
16     SOMD_Init(__SOMEnv);
17
18     Attrib *remoteObj =
19         (Attrib *)somdCreate(__SOMEnv, "Attrib", TRUE);
20     assert(! checkError(__SOMEnv, TRUE));
21
22     cout << "Remote object i is " << remoteObj->i << endl;
23     remoteObj->i = 5;
24     cout << "Remote object i is " << remoteObj->i << endl;
25
26     remoteObj->somFree();

```

```

27
28     SOMD_Uninit(__SOMEnv);
29     SOM_DestroyLocalEnvironment(__SOMEnv);
30
31     return(0);
32 }

```

Direct Data Access

Making all instance data members into attributes handles most of the direct data access situations that can occur from the client. Data will be accessed directly only in the class implementation code, where private data access is valid. However, there are situations where direct access, even to private data, is invalid.

Consider the situation shown in Figure 8.4, where the object `remoteObj1` and `remoteObj2` are created through different server processes. In this situation, an expression such as `remoteObj1=remoteObj2` would result in the assignment method for `remoteObj1` being invoked, with `remoteObj2` as a parameter. Since the assignment method is part of the class implementation, it has access to the private data of both objects (because they are the same type). Thus the assignment method, running in process 1 for `remoteObj1`, would attempt to directly access the private data of `remoteObj2` in order to copy the instance data from `remoteObj2` to `remoteObj1`. Direct data access in this case is invalid, however, because the instance data of `remoteObj2` is local to process 2, not process 1.

Hello Application

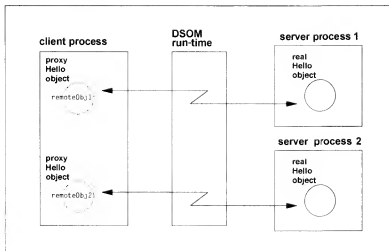


FIGURE 8.4 Creating objects of same type in different servers.

For such situations, in addition to making all instance data into attributes, you must also use the **SOMNoDataDirect** pragma to indicate that the compiler should not use direct data access for any access to an object except through the **this** pointer. Data is accessed through the **this** pointer only in the server process on which the object was created. Restricting direct data access to the **this** pointer ensures that direct data access does not take place except where the data is allocated locally.

The simplest way to prevent invalid direct data access is to put **#pragma SOMNoDataDirect(*)** inside the class definition. This will ensure that the **_get** and **_set** data access methods will be used for all data access except that explicitly through the **this** pointer. On the client side, this will be all data access, and on the server side, it will be any access except through the **this** pointer.

Specifying **SOMNoDataDirect** in the class definition will also require that you make all data members into attributes, so that they can be accessed through the **_get/_set** methods. But if you will always be creating objects of a given class through the same server, you can improve performance and avoid making private data members into attributes by specifying only **#pragma SOMNoDataDirect(on)** at the client side. This will allow all implementation code to access the data directly. But you must ensure that remote objects of a given class are created only through a single server so that their instance data is always local to that server.

As discussed in Chapter 4, *Using DirectToSOM C++*, recall that it is possible to make the backing data for an attribute public, using the **publicdata** keyword with the **SOMAttribute** pragma, in which case, the compiler would attempt to directly access data on the client side also. You can avoid this causing problems with direct data access by always setting direct data access off for any classes with which you will be creating remote objects. You can do this by putting **#pragma SOMNoDataDirect(on)** and **#pragma SOMNoDataDirect(off)** in the client code around the inclusion of the header file for such classes. Alternatively, you can compile your client code with a compiler switch that turns **SOMNoDataDirect** on for the entire compilation (for example, **/Gb+** for OS/2 and Windows, and **-qdsom** for AIX).

To summarize the data access guidelines when your class will be used through DSOM, you should make sure that you have done the following:

1. Make all public data members into attributes and specify **SOMNoDataDirect** in the client code for all remote classes.
2. If objects of a derived class and its base class will be created by different servers, make all base class protected data members into attributes and specify **SOMNoDataDirect** in the derived class implementation code.
3. If objects of the same class will be created by different servers, make all base class private data members into attributes and specify **SOMNoDataDirect** in the class implementation code.

Memory Management

DSOM supports remote method calls by copying parameter data from the client address space to the server address space. For **in** parameters, only a copy to the server occurs; for **inout** parameters, a copy to and from the server is made; and for **out** parameters and method return values, only a copy from the server is made. For **in** or **inout** parameters, DSOM allocates storage for the underlying type at the server, copies the incoming data to the server address space, and passes the address of this storage to the receiving server method.

There are several choices for handling memory management in DSOM: **caller-owned**, **object-owned**, and **dual-owned**. These options affect when storage is deleted on both the client and server side, and whether the client application, the object implementation, or the DSOM run time is responsible for owning the storage. The simplest way to handle memory management with DSOM and DirectToSOM C++ is to use the DSOM 3.0 default of **caller-owned**, in which DSOM assumes that all storage is owned by the client application, and allocates/deallocates memory in such a way as to transfer ownership to the client application. The other two options are **object-owned**, in which the object implementation is responsible for handling memory management, and **dual-owned**, which is like caller-owned on the client application side and object-owned on the object implementation side.

Currently for DSOM 3.0, the default of **caller-owned** memory management is in effect only for classes built using the C or C++ bindings. This is due to the marshaling information generated within the class binary for the bindings (with a default of **caller-owned** memory management) that allows DSOM to access class information without using the interface repository. This support is not yet available for DirectToSOM C++, so you must specify the IDL modifier **memory_management=corba** with the class definition using the **SOMIDLPass** pragma in order to get **caller-owned** semantics:

```
#pragma SOMIDLPass(*, "Implementation-Begin", "memory_management = corba;")
```

When DirectToSOM C++ support is available for accessing DSOM information through the class binary, this modifier will no longer be necessary.

Figure 8.5 details how **caller-owned** memory management works. The labels above each box indicate the allocator of that memory. As shown in the top panel, the client application is responsible for allocating and initializing storage for **in** and **inout** parameters (and for initializing the pointer used for **out** parameters). DSOM allocates corresponding storage on the server side and copies the input values across. The middle panel shows that the object implementation on the server side is responsible for allocating storage for the **out**, **return**, and, optionally, the **inout** parameter results.

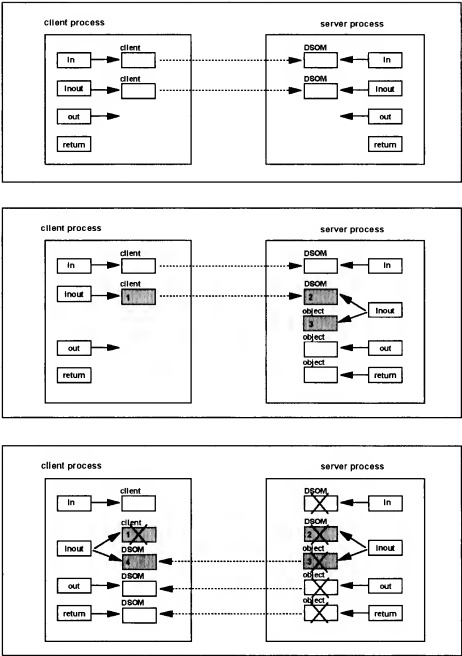


FIGURE 8.5 Caller-owned memory management.

In the third panel, DSOM allocates storage on the client side corresponding to the returned **out**, **return**, and **inout** parameter results and optionally deallocates the client's storage for the **inout** parameter input value. In addition, DSOM will implicitly deallocate the storage on the server side for **in**, **out**, **inout**, and pointer type **return** values, as part of transferring ownership of that storage to the client application. The storage marked with an X is implicitly deallocated by DSOM. Because DSOM deallocates the storage on the server side, it implies that the object implementation cannot directly return static storage for pointer types—it must return storage that can be deallocated by DSOM.

inout parameters are handled differently depending upon the object implementation and the parameter type. At the server process, DSOM always deallocates the storage currently referenced by the **inout** parameter at the server side upon return to the client. If the object implementation reuses the storage provided by DSOM (shaded box 2) to return the result, DSOM will deallocate that storage upon returning. If the object implementation provides a different memory address for the **inout** parameter result, then DSOM will deallocate the storage in both boxes 2 and 3.

At the client side, DSOM will reuse original **inout** parameter storage (box 1) for scalar types and strings where the result is less than or equal to the original parameter size. For strings results that are longer than the initial input value, or object types, the DSOM run time will deallocate the original storage (box 1) and will allocate new storage (box 4). In addition, if a null pointer is returned, DSOM will also deallocate the original storage and set the **inout** parameter value to null (no new storage is allocated). You can prevent DSOM from deallocating the caller's original storage using the **suppress_inout_free** IDL modifier. However, this can result in unreachable storage if the caller's original storage was dynamically allocated and the address was not saved.

When local object references are being returned from a server process to a client process as **out**, **inout**, or **return** results, DSOM does not deallocate the local object being returned. Instead, DSOM implicitly creates a corresponding **SOMDObject** for the local object, which will be marshaled and released on return from the method instead of the local object itself. If, however, the object being returned is a proxy for an object in another process, then DSOM deallocates that proxy object on return from the method, as described previously.

Figure 8.6 illustrates **object-owned** memory management. In this situation, storage is owned by the object implementation and the client application cannot deallocate it. The first two panels are the same as caller-owned—DSOM allocates storage on the server side for the incoming **in** and **inout** parameter values, and the object implementation is responsible for allocating storage for the **out**, **return**, and, optionally, **inout** parameter results. The third panel illustrates the key difference. With object-owned memory management, DSOM implicitly deallocates only the client storage for **in** parame-

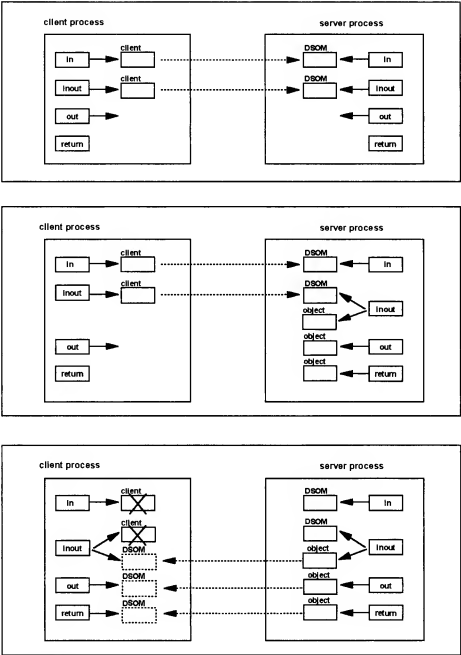


FIGURE 8.6 Object-owned memory management.

ters and for **inout** parameters whose storage will not be reused. The ownership of this storage is essentially transferred to the object implementation. Note also that the storage allocated by DSOM in the client side for the **out**, **return**, and **inout** results is owned by the object implementation—the client cannot explicitly deallocate this storage (this is why that storage is shown with a dashed line in the client process). The object implementation is also given ownership of reused **inout** parameter storage. The client-side storage will be deallocated when the client process deallocates the proxy, or when the **somdReleaseResources** method is invoked against the proxy.

Dual-owned memory management, shown in Figure 8.7, is a combination of **caller-owned** and **object-owned**. Both the client and the remote object implementation retain ownership of object storage, and DSOM only deallocates storage for client-side **inout** storage that is not reused. The caller is responsible for deallocating storage on the client side, while the object implementation is responsible for deallocating storage on the server side.

Client Memory Management

With **caller-owned** memory management, the caller is always responsible for freeing any data that was allocated as a result of a method invocation. In most cases the client is responsible for allocating storage for parameters, too. But there are several cases where the object implementation must allocate storage for values returned from the server. With **caller-owned** memory management, this memory will be transferred to the caller by DSOM, and it is the responsibility of the client to deallocate this storage. Strictly from a DirectToSOM C++ perspective (there are IDL-specific types that have further restrictions), these cases are:

- ♦ `char *` (or **string**) when used as **out** arguments or **return** results
- ♦ pointer types (other than `char *`) and objects when used as **inout** or **out** arguments or as **return** results
- ♦ arrays when used as **out** arguments or **return** results. For **out** array arguments, the caller is responsible for allocating the top-level array storage and the target object implementation will allocate the nest. For example, if you are passing an array of integers as **out** the caller allocates the entire array and the object implementation simply fills it. But for an array of strings the object implementation will allocate storage for each string. The ownership of the storage for each string will be transferred to the caller, which must deallocate that storage. For an example of passing an array as **out** see the *Parameter Passing* section later in this chapter. For array **return** results, the object implementation will allocate the entire array.

Note that **inout** parameters may potentially be reallocated by DSOM on the client side, so you cannot rely on accessing the original address to obtain the output result. This reallocation will take place for strings and sequences

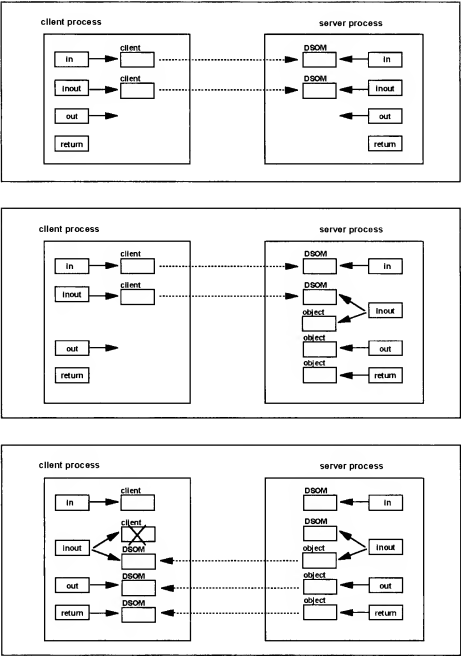


FIGURE 8.7 Dual-owned memory management.

where the output value size is larger than the input value and for all object and **SOMFOREIGN** parameter types (also for all **any** or **TypeCode** parameters, but you are not likely to encounter such types when working with DirectToSOM C++). This reallocation may take place in cases where the output value is larger than the input value. In addition, if a pointer type or string output value is null, DSOM will deallocate the original storage.

You must therefore be careful that any data passed as **inout** is dynamically allocated so that it can be deallocated, which implies that data passed as an **inout** parameter cannot be static or automatic. The alternative is to prevent this behavior using the **suppress_inout_free** modifier, which can be applied to a method by listing the particular parameters to which it should apply. DSOM may still allocate new storage to hold the resulting value, so if you do pass dynamically allocated storage and specify **suppress_inout_free**, the original storage may become unreachable if you don't save the address.

The next example shows the definition of the class `Memory` that has a single method, `StringParams`. **memory_management = corba** is specified with the **SOMIDLPass** pragma at lines 7 and 8. The `StringParams` method accepts two **inout** string parameters. The first parameter, `arg1`, is designated with **suppress_inout_free**, which indicates that DSOM should not deallocate the caller's original passed storage on return from a remote method invocation. (I have also used the **SOMIDLDecl** pragma to ensure that the IDL argument name `arg1`, generated into the IDL by the compiler, will match the one I've used for **suppress_inout_free**. Currently, there is no documented convention for DirectToSOM C++ compiler-generated IDL parameter names, so using **SOMIDLDecl** will prevent a compiler change in the argument names from invalidating the **suppress_inout_free** modifier, and will allow the program to be portable between DirectToSOM C++ compilers.)

In the implementation for the class, the method `StringParams` allocates storage for the return values. DSOM will deallocate this storage (on the server side only) on return from the method invocation, following the semantics of **caller-owned** memory management.

`tstmem.cpp` shows how the class would be used. Lines 18 through 24 allocate storage for two strings, `str1` and `str2`, and copy the value of the array `txt` into these strings. Because the first parameter is designated as **suppress_inout_free**, the storage address for `str1` must be saved prior to making the remote method call, otherwise the storage will become unreachable.

The remote method call to `StringParams` is made at line 33. Following the invocation of this method, new storage will be allocated for both return strings, because they are longer than the original strings. The original storage for `str2` will be implicitly deallocated by DSOM, but that of `str1` will remain allocated, due to the **suppress_inout_free** modifier. So, at line 41, the original storage for `str1` is deleted, using the saved address.

The storage for `str1` and `str2` will have been reallocated by DSOM using **SOMMalloc**, so we cannot simply deallocate them using **delete**,

because **SOMMalloc** and **new** are different storage allocators. If the strings were reallocated, they must be deallocated using **SOMFree**. If the storage was not reallocated, **delete** must be used. Line 43 checks if `p1` is equal to `str1`. If they are equal, no further action is necessary, because the storage will have already been deallocated at line 41. Otherwise, the storage, which was allocated by DSOM, is deallocated using **somFree**. Lines 47 through 52 perform similar checking for `p2` and `str2`.

Determining which storage allocator to use can be quite cumbersome, and in some cases quite difficult. But there is a solution, which is discussed later in this chapter. The output for this program, run in OS/2 is:

```
str1: 0xd32fd0, This is a short string
str2: 0xd32f50, This is a short string
str1: 0xdba290, This is a long string that is at least 20 characters
str2: 0xdba710, This is a long string that is at least 20 characters
```

You can see that the addresses contained in `str1` and `str2` are different after the remote method call to `StringParms`. This is because the **inout** result was larger than the original input value, so DSOM allocated new storage to hold the larger return value.

Definition of DirectToSOM C++ Class Memory (suppress\memory.hh):

```
1  #include <som.h>
2
3  class Memory : public SOMObject {
4      #pragma SOMClassName(*, "Memory")
5      #pragma SOMIDLPass(*, "Implementation-End", \
6                          "dllname = \"memory.dll\";")
7      #pragma SOMIDLPass(*, "Implementation-Begin", \
8                          "memory_management = corba;")
9  public:
10     void StringParms(char **, char **);
11     #pragma SOMMethodName(StringParms, "StringParms")
12     #pragma SOMIDLDecl(StringParms, \
13                         "void StringParms(inout string arg1, \" \
14                                     \"inout string arg2)\")
15     // following prevents deallocation of first argument
16     #pragma SOMIDLPass(*, "Implementation-End", \
17                         "StringParms : suppress_inout_free = arg1;")
18 };
```

Implementation of DirectToSOM C++ Class Memory (suppress\memory.cpp):

```
1  #include <iostream.h>
2  #include <assert.h>
3  #include "memory.hh"
4
```

```

5 void Memory::StringParms(char **str1, char **str2)
6 {
7     char longString[] =
8         "This is a long string that is at least "
9         "20 characters";
10
11     // DSOM deallocates on return
12     *str1 = new char[sizeof(longString)];
13     assert(*str1);
14     strcpy(*str1, longString);
15
16     // DSOM deallocates on return
17     *str2 = new char[sizeof(longString)];
18     assert(*str2);
19     strcpy(*str2, longString);
20 }

```

Client of DirectToSOM C++ Class Hello (suppress\tstmem.cpp):

```

1 #include <iostream.h>
2 #include <assert.h>
3 #include <somd.hh>
4 #include "check.h"
5 #include "memory.hh"
6
7 int main(int argc, char *argv[])
8 {
9     __SOMEnv = SOM_CreateLocalEnvironment();
10
11     SOMD_Init(__SOMEnv);
12
13     // create remote object and assign proxy to remoteObj
14     Memory *remoteObj =
15         (Memory *)somdCreate(__SOMEnv, "Memory", TRUE);
16     assert(! checkError(__SOMEnv, TRUE));
17
18     char txt[] = "This is a short string";
19     char *str1 = new char[sizeof(txt)];
20     assert(str1);
21     strcpy(str1, txt);
22     char *str2 = new char[sizeof(txt)];
23     assert(str2);
24     strcpy(str2, txt);
25
26     cout << "str1: " << (void *)str1 << ", "
27         << str1 << endl;
28     cout << "str2: " << (void *) str2 << ", "
29         << str2 << endl;
30
31     // save address so can free later
32     char *p1 = str1, *p2 = str2;
33     remoteObj->StringParms(&str1, &str2);
34

```

Continued

```

35     cout << "str1: " << (void *)str1 << ", "
36         << str1 << endl;
37     cout << "str2: " << (void *)str2 << ", "
38         << str2 << endl;
39
40     // suppress_inout_free didn't delete
41     delete p1;
42
43     if (p1 != str1)
44         // reallocated by DSOM using SOMMalloc
45         SOMFree(str1);
46
47     if (p2 != str2)
48         // reallocated by DSOM using SOMMalloc
49         SOMFree(str2);
50     else
51         // original storage allocated by new
52         delete str2;
53
54     // delete proxy and remote object
55     remoteObj->somFree();
56
57     SOMD_Uninit(__SOMEnv);
58     SOM_DestroyLocalEnvironment(__SOMEnv);
59
60     return(0);
61 }

```

Server Memory Management

Caller-owned memory management requires that DSOM transfer ownership of all server-allocated parameter memory to the client side, to avoid potential memory leaks. The result is that DSOM deallocates all parameter memory on the server side as part of returning a response to the client. For **in** and **inout** parameters, where DSOM has allocated the storage already, this does not present a problem. But, for **out** or **return** results, or **inout** parameters where the object is returning different storage than that allocated by DSOM, this implies that the storage returned cannot be static; that is, it must have been dynamically allocated by the object so that DSOM can deallocate it.

The DSOM memory management policy presents a problem for the compiler-generated **_get** and **_set** routines for pointer type or object attributes, which do not conform to the **caller-owned** memory management policy. The **_get** method simply returns the address of the underlying data, which DSOM will deallocate as part of the return semantics, so the next time you access that data, it will no longer be valid. For example, if you request the value of a **string** attribute, on the return from the default **_get** method, the **string** storage at the server side will be implicitly deallocated by DSOM.

You have several choices to prevent this situation: designate the attribute **_get** method memory management as either **dual-owned** or **object-owned** for the return value, or supply a **_get** method that will explic-

itly allocate storage, copy the output value to that storage, and return that storage (which will be deallocated by DSOM).

Specifying **dual-owned** memory management for the return value makes it unnecessary to explicitly supply a `_get` method, and it will result in **caller-owned** behavior at the client side, which is consistent with other methods. And on the server side, the object is responsible for ownership of the data, not the client/DSOM, so the storage will not be inadvertently deallocated. The disadvantage of this approach is that you must explicitly retrieve the storage and release it on the client side, which makes it a little cumbersome to use the attribute in an expression. Further, this approach is not local/remote transparent—you have to write your code for local calls differently from remote calls.

Using **object-owned** memory management is preferable, because the storage will not be deallocated on the server side and the caller side does not have to retrieve the storage and release it explicitly; and you don't need to provide a `_get` method. To avoid a large amount of memory consumption, however, you should periodically issue the `somdReleaseResources` method against the proxy on the client side.

The default `_set` method also requires some attention. The server side copy of any input parameter storage (that is, for **in** or **inout** parameters) will be deleted after the method call with **caller-owned** memory management. Therefore, if you need to save any parameters values on the server side, you will need to copy the values to storage allocated explicitly in the server. In other words, if the method receives the address of an integer, you cannot simply store that address, you must allocate new storage for that integer and save it there. This presents a problem with the default `_set` method for pointer type and object attributes.

Unfortunately, you cannot simply designate the input parameter as **object-owned**, as for the `_get` method. While this would prevent the storage from being deallocated, it could also cause memory leaks because each time the attribute value was updated, the address of the previously allocated storage would be overwritten. So for `_set` methods for pointer types or objects, you must explicitly provide a `_set` method that deallocates any existing storage prior to setting the attribute value. However, this only solves part of the problem, as DSOM will still deallocate the input parameter storage on the server side following **caller-owned** memory management semantics. So you must either allocate storage and copy the input parameter value to that storage in the `_set` method or designate the memory management as **object-owned** or **dual-owned**. In this case, specifying **dual-owned** is the preferable approach, as it avoids the extra storage allocation and copy, and prevents the original client-side input parameter storage from being deallocated, which is more natural. It does not provide local/remote transparency, though. If the class will be used both locally and remotely, you should consider using **object-owned** instead.

The following illustrates how to define a **string** attribute with proper memory management. The string `txt` is defined as an attribute with a user-supplied `_set` method at line 11 in the file `attrib.hh`. Using the **SOMIDLPass** pragma, the modifier **dual_ownership_parameters** is specified for the `_set_text` method, while **object_owns_results** is specified for `_get_text`.

In the class implementation, the user-supplied `_set_text` method, at lines 10 through 17, checks for existing storage, in which case it is deleted. Then the string argument is assigned to the attribute value at line 16. The client program invokes the `_set/_get` methods several times to ensure that memory management is being handled correctly.

Definition of DirectToSOM C++ Class *Attrib* (*attrmem\attrib.hh*):

```

1  #include <som.hh>
2
3  class Attrib : public SOMObject {
4      #pragma SOMClassName(*, "Attrib")
5      #pragma SOMIDLPass(*, "Implementation-End", \
6          "dllname = \"attrib.dll\";")
7      #pragma SOMIDLPass(*, "Implementation-Begin", \
8          "memory_management = corba;")
9  public:
10     char *txt;
11     #pragma SOMAttribute(txt, noset)
12     #pragma SOMIDLPass(*, "Implementation-End", \
13         "_set_txt : dual_ownership_parameters = txt;")
14     #pragma SOMIDLPass(*, "Implementation-End", \
15         "_get_txt : object_owns_result;")
16     Attrib();
17 };

```

Implementation of DirectToSOM C++ Class *Attrib* (*attrmem\attrib.cpp*):

```

1  #include <fstream.h>
2  #include <assert.h>
3  #include "attrib.hh"
4
5  Attrib::Attrib()
6  {
7      txt = NULL;
8  }
9
10 void Attrib::_set_txt(char *str)
11 {
12     if (txt) {
13         cout << "deleting text: " << txt << endl;
14         delete txt;

```

```

15     }
16     txt = str;
17 }

```

Client of DirectToSOM C++ Class *Attrib* (*attrmem\stattrib.cpp*):

```

1  #include <iostream.h>
2  #include <assert.h>
3  #include <somd.hh>
4  #include 'check.h'
5  #pragma SOMNoDataDirect(on)
6  #include "attrib.hh"
7  #pragma SOMNoDataDirect(off)
8
9  int main(int argc, char *argv[])
10 {
11     __SOMEnv = SOM_CreateLocalEnvironment();
12
13     // Initialize DSOM run-time environment
14     SOMD_Init(__SOMEnv);
15     // Handle memory dealloc with delete/SOMFree
16     SOMD_NoORBFfree();
17
18     Attrib *remoteObj =
19         (Attrib *)SOMD_ObjectMgr->somdNewObject("Attrib", "");
20     assert(! checkError(__SOMEnv, TRUE));
21
22     cout << "setting to hello" << endl;
23     remoteObj->txt = "Hello";
24     cout << "Remote object is " << remoteObj->txt << endl;
25     cout << "Remote object is " << remoteObj->txt << endl;
26     cout << "setting to Hello2" << endl;
27     remoteObj->txt = "Hello2";
28     cout << "Remote object is " << remoteObj->txt << endl;
29     cout << "Remote object is " << remoteObj->txt << endl;
30     ((SOMDClientProxy *)remoteObj)->somdReleaseResources();
31
32     remoteObj->somFree();
33     SOMD_Uninit(__SOMEnv);
34     SOM_DestroyLocalEnvironment(__SOMEnv);
35
36     return(0);
37 }

```

Shallow vs. Deep Copy

Be aware when copying structures containing embedded pointers that you will need to do a *deep* copy, rather than a *shallow* copy. For example, suppose a structure that contains a string is passed as a parameter through DSOM to a method. If that target method wants to keep a copy of the struc-

ture, it must copy not only the top-level, but each substructure also. In this example, the target method would need to allocate storage for the embedded string and copy the string contents too.

ORBFree vs. SOMFree vs. delete

When DSOM allocates memory, it uses its own memory allocation routines in which case to deallocate the storage you need to use **ORBFree**, rather than **SOMFree**. It can be difficult, if not impossible, to determine which memory management routine to call based on how the storage was allocated. But, fortunately, there is a solution.

The main difference between **ORBFree** and **SOMFree** is that **ORBFree** will free any memory embedded in the storage that you are freeing, whereas **SOMFree** just deallocates the top-level storage (like **free** in C). But, we can use destructors for that in C++, and avoid having to deal with **ORBFree** altogether. The way to do this is insert a call to **SOMD_NoORBfree** at the start of client program. This indicates that **ORBFree** will not be used to deallocate storage, but **SOMFree** will be used instead. (Note that **ORBFree** does not apply to **inout** storage, which is why I used **SOMFree** in the earlier example of **suppress_inout_free**.)

The second part of the solution is to make sure that all your memory is being allocated by the same allocator. We solved the first problem of not having to call **ORBFree**, but we still need to know if memory was allocated by SOM or by the C++ memory allocation routines in order to determine which deletion routine to call. The problem occurs with types that are not SOM classes, such as strings.

Normally, when **new** and **delete** are called, the default C++ memory allocation routines are used. For SOM classes, as discussed in Chapter 5, **new** and **delete** are mapped to **SOMMalloc** and **SOMFree**. However, if, for example, DSOM allocates storage for a returned string, it will use **SOMMalloc**. You cannot subsequently delete this storage using **delete**, because it was not allocated with the C++ memory allocator. As with **ORBFree**, having to determine when to call **delete** and when to call **SOMFree** for non-SOM classes can be very difficult. You really want to be able to blindly use **new** and **delete** for everything.

Fortunately, there is a way to do this. Actually, there are two ways: you can either map the C++ memory allocation to the SOM memory allocation routines, or vice-versa. Initially, the best approach is the former, because the SOM memory allocation routines perform additional error handling that could be useful in debugging your application. Later, you may want to consider mapping the SOM routines to the C++ routines, only for performance reasons. This will avoid the additional overhead of the SOM error handling. Note, however, that using the C++ memory management routines may preclude sharing memory in the same process with any SOM program that was not compiled with the same C++ compiler. Other such programs

can use the memory, but they cannot deallocate it or allocate memory that your program can deallocate, as the memory allocators will be different. Using the SOM memory allocation routines ensures a common allocator for all programs.

Mapping all memory management to the same allocators, together with invoking **SOMD_NoORBfree**, will keep the memory management headaches of your application to a minimum.

The following example shows how to map the C++ memory allocation routines to the SOM allocation routines. By compiling and linking the object into your application, as shown in the makefile, all C++ memory allocation will implicitly use the SOM routines, because user-defined functions take precedence over library-supplied ones.

As shown in the sample program, you can use **new** and **delete** for everything, because all storage allocation, either explicitly in your program or implicitly by DSOM, will use the SOM memory allocation routines. (You may need to specify a compiler option to avoid duplicate symbol errors with the library-supplied memory management routines when linking this file with your program. For OS/2 and Windows, the option is **/B"/NOE**".)

Map C++ Memory Allocation to SOM Memory Allocation (**memory1/sommem1.cpp**):

```

1  #include <stdlib.h>
2  #include <assert.h>
3  #include <som.hh>
4
5  void* operator new(size_t size)
6  {
7      void *ptr = SOMMalloc(size);
8      assert(ptr);
9      return ptr;
10 }
11
12 void* operator new[](size_t size)
13 {
14     void *ptr = SOMMalloc(size);
15     assert(ptr);
16     return ptr;
17 }
18
19 void operator delete(void *ptr)
20 {
21     SOMFree(ptr);
22 }
23
24 void operator delete[](void *ptr)
25 {
26     SOMFree(ptr);
27 }
```

Using Common Memory Allocation (memory1/tstmem.cpp):

```

1  #include <iostream.h>
2  #include <assert.h>
3  #include <somd.hh>
4  #include "check.h"
5  #include "memory.hh"
6
7  int main(int argc, char *argv[])
8  {
9      __SOMEnv = SOM_CreateLocalEnvironment();
10
11     SOMD_Init(__SOMEnv);
12
13     Memory *remoteObj =
14         (Memory *)somdCreate(__SOMEnv, "Memory", TRUE);
15     assert(! checkError(__SOMEnv, TRUE));
16
17     char txt[] = "This is a short string";
18     char *str1 = new char[sizeof(txt)];
19     assert(str1);
20     strcpy(str1, txt);
21     char *str2 = new char[sizeof(txt)];
22     assert(str2);
23     strcpy(str2, txt);
24
25     cout << "str1: " << (void *)str1 << ", "
26           << str1 << endl;
27     cout << "str2: " << (void *)str2 << ", "
28           << str2 << endl;
29
30     // save address so can free later
31     char *p1 = str1, *p2 = str2;
32     remoteObj->StringParms(&str1, &str2);
33
34     cout << "str1: " << (void *) str1 << ", "
35           << str1 << endl;
36     cout << "str2: " << (void *)str2 << ", "
37           << str2 << endl;
38
39     // p1 was allocated by client, but str1
40     // and str2 were potentially reallocated by DSOM,
41     // so must ensure that use correct free function
42     if (p1 != str1)
43         // must delete because suppress_inout_free didn't
44         delete p1;
45     delete str1;
46     delete str2;
47
48     // delete proxy and remote object
49     remoteObj->somFree();
50
51     SOMD_Uninit(__SOMEnv);
52     SOM_DestroyLocalEnvironment(__SOMEnv);

```

```

53
54     return(0);
55 }

```

Makefile (memory1/makefile):

```

1  all: sommem1.obj memory.dll tstmem.exe somdimpl.dat
2
3  ICCOPTS = /I ..\include
4
5  memory.dll: memory.cpp memory.hh memoryi.c
6             gcc $(ICCOPTS) /Ti+ /Ge- /B"/NOE" memory.cpp \
7             memoryi.c memory.def sommem1.obj
8             implib memory.lib memory.dll
9
10 tstmem.exe: tstmem.cpp memory.hh
11            gcc $(ICCOPTS) /Ti+ /B"/NOE" \
12            tstmem.cpp memory.lib sommem1.obj
13
14 sommem1.obj: sommem1.cpp
15            gcc $(ICCOPTS) /Ti+ -c sommem1.cpp
16
17 memory.idl: memory.hh
18            gcc $(ICCOPTS) memory.hh
19            sc -sir -u memory.idl
20
21 memoryi.c: memory.idl
22            sc -simod memory.idl
23            sc -sdef memory.idl
24
25 somdimpl.dat:
26            regimpl -D -i MemoryServer
27            regimpl -A -i MemoryServer
28            regimpl -a -i MemoryServer -c Memory

```

The second method is to map the SOM memory allocation routines to those of C++. You can only do this from a program executable (as opposed to a DLL), in order to ensure that the new memory management routines are set up prior to any SOM memory being allocated. On the client side, you can simply compile and link the program file that follows with the application.

The file contains a definition for the class **SOMMemory**, the constructor for which switches the SOM memory allocation routines to the functions supplied in the file, which call the C memory allocation routines. The four SOM memory allocation functions **SOMMalloc**, **SOMCalloc**, **SOMRealloc**, and **SOMFree** are actually global function pointers that can be updated to supply different memory management routines. The priority for the subsequent static objects in the file is set to -1000 using the **priority** pragma, which ensures that static objects in this file will be initialized before any others. The static object **dummy** of type **SOMMemory** is then declared at line 23. When a program linked to this file is loaded, the static initialization for **dummy**

will cause the memory allocation routines to be switched prior to any other user static initialization code.

Because you can only change the SOM memory routines from an executable, in order to switch memory on the server side, you must provide your own server program and link the new memory management routines to this program. See the *Customizing the Server Program* section later in this chapter for details. In addition, you must ensure that the C++ libraries are linked in dynamically rather than statically (in OS/2 or Windows, use the compiler option */Gd+*). This is required to prevent the C++ libraries from being terminated before the SOM libraries, because switching the memory management routines causes the SOM library termination to depend upon storage in the C++ heap.

Map SOM Memory Allocation to C++ Memory Allocation (memory2/sommem2.cpp):

```

1  #include <stdlib.h>
2  #include <limits.h>
3  #include <assert.h>
4  #include <som.h>
5
6  void* SOMLINK mySOMMalloc(size_t size);
7  void* SOMLINK mySOMCalloc(size_t nobj, size_t size);
8  void* SOMLINK mySOMRealloc(void *ptr, size_t size);
9  void SOMLINK mySOMFree(void *ptr);
10
11 class SOMMemory {
12     void* (SOMLINK *saveSOMMalloc) (size_t);
13     void* (SOMLINK *saveSOMCalloc) (size_t, size_t);
14     void* (SOMLINK *saveSOMRealloc) (void *, size_t);
15     void (SOMLINK *saveSOMFree) (void *);
16 public:
17     SOMMemory();
18     ~SOMMemory();
19 };
20
21 #pragma priority(-1000) // ensures is initialized first
22
23 static SOMMemory dummy;
24
25 SOMMemory::SOMMemory()
26 {
27     saveSOMMalloc = SOMMalloc;
28     SOMMalloc = mySOMMalloc;
29     saveSOMCalloc = SOMCalloc;
30     SOMCalloc = mySOMCalloc;
31     saveSOMRealloc = SOMRealloc;
32     SOMRealloc = mySOMRealloc;
33     saveSOMFree = SOMFree;
34     SOMFree = mySOMFree;
35 }
```



```

36
37 SOMMemory::~SOMMemory()
38 {
39     SOMMalloc = saveSOMMalloc;
40     SOMCalloc = saveSOMCalloc;
41     SOMRealloc = saveSOMRealloc;
42     SOMFree = saveSOMFree;
43 }
44
45 void* SOMLINK mySOMMalloc(size_t size)
46 {
47     void *ptr = malloc(size);
48     assert(ptr);
49     return ptr;
50 }
51
52 void* SOMLINK mySOMCalloc(size_t nobj, size_t size)
53 {
54     void *ptr = calloc(nobj, size);
55     assert(ptr);
56     return ptr;
57 }
58
59 void* SOMLINK mySOMRealloc(void *ptr, size_t size)
60 {
61     ptr = realloc(ptr, size);
62     assert(ptr);
63     return ptr;
64 }
65
66 void SOMLINK mySOMFree(void *ptr)
67 {
68     free(ptr);
69 }

```

Makefile (memory2/makefile):

```

1 all: sommem2.obj memory.dll tstmem.exe myserver.exe somdimpl.dat
2
3 ICCOPTS = /I ..\include
4
5 memory.dll: memory.cpp memory.hh memoryi.c
6     icc $(ICCOPTS) /Ti+ /Ge- /Gd+ /B"/NOE" \ memory.cpp memoryi.c
7         memory.def
8
9     implib memory.lib memory.dll
10
11 tstmem.exe: tstmem.cpp memory.hh
12     icc $(ICCOPTS) /Ti+ /Gd+ tstmem.cpp \ memory.lib sommem2.obj
13
14 myserver.exe: myserver.cpp
15     icc $(ICCOPTS) /Ti+ /Gd+ myserver.cpp scmmem2.obj
16

```

Continued

```

17 sommem2.obj: sommem2.cpp
18     icc $(ICCOPTS) /Ti+ -c sommem2.cpp
19
20 memory.idl: memory.hh
21     icc $(ICCOPTS) memory.hh
22     sc -sir -u memory.idl
23
24 memoryi.c: memory.idl
25     sc -simod memory.idl
26     sc -sdef memory.idl
27
28 sondimpl.dat:
29     regimpl -D -i MemoryServer
30     regimpl -A -i MemoryServer -p myserver.exe
31     regimpl -a -i MemoryServer -c Memory

```

Parameter Passing

This section describes various programming considerations and techniques for passing parameters with DSOM.

Parameter Directional Attributes

If you are passing a parameter such as a pointer type and expect the server to return data back to the client through that parameter, make sure the IDL has **inout**, or force the IDL to **out**. If the parameter is defined as **in**, data will not be returned, due to the marshaling of parameters and how memory management is performed. Specifically, when you want to receive a string, as for `char *` types, which are mapped to the **string** type in IDL, there are several options:

- ♦ Define the parameter as `char **`, which will map to the IDL parameter type **inout string**. Don't forget that if the return string is larger than the input string, DSOM will deallocate the original string and allocate storage for the larger string on the client side. If this is a possibility, ensure that the input string was dynamically allocated. (For DSOM 2.x, this reallocation will not occur—the output result will be truncated to the size of the original input string.)
- ♦ Define the parameter as `char **` and force the IDL to be an **out** parameter using the **SOMIDLDecl** pragma (there is no other way in DirectToSOM C++ to specify generation of an **out** parameter). The easiest way to do this is by looking at the generated IDL, which will have a **inout string** parameter, and specifying this in the **SOMIDLDecl** but using **out** for the parameter. I also assign a specific name **SOMMethodName** to circumvent the mangling of the function name, which will differ depending on the parameters being passed. (See Chapter 7 for an example.)
- ♦ Return the string as a `char *`.

Passing Arrays

For passing and returning arrays it is best to declare the array explicitly, rather than just as a pointer. Earlier versions of the compiler did not generate complete array information in the IDL, so verify that the IDL contains the array correctly in the parameter list. If only a pointer is generated, DSOM will marshal just the first element of the array. The following example shows how to verify that an array is defined correctly in the generated IDL; it includes appropriate memory management handling for the methods. Note that the caller supplies the top-level array storage for the array parameter, which is filled in by the `getArray` implementation. `getArray` does not need to explicitly allocate storage for the array.

Definition of DirectToSOM C++ Class Array (array\array.hh):

```

1  #include <som.hh>
2
3  #define ARR_SIZE      10
4  #pragma SOMNoMangling(on)
5
6  class Array : public SOMObject {
7      #pragma SOMClassName(*, "Array")
8      #pragma SOMIDLPass(*, "Implementation-End", \
9          "dllname = \"array.dll\";")
10     #pragma SOMIDLPass(*, "Implementation-Begin", \
11         "memory_management = corba;")
12     int intArray[ARR_SIZE];
13     public:
14         void setArray(int in[ARR_SIZE]);
15         #pragma SOMIDLDecl(setArray, \
16             "void setArray(in long arg1[10])")
17         void getArray(int in[ARR_SIZE]);
18         #pragma SOMIDLDecl(getArray, \
19             "void getArray(out long arg1[10])")
20 };

```

Implementation of DirectToSOM C++ Class Array (array\array.cpp):

```

1  #include <fstream.h>
2  #include <assert.h>
3  #include "array.hh"
4
5  void Array::setArray(int in[ARR_SIZE])
6  {
7      cout << "Array::passArray in: ";
8      for (int i=0; i < ARR_SIZE; i++)
9          cout << (intArray[i] = in[i]) << " ";
10     cout << endl;
11 }
12
13 void Array::getArray(int outArr[ARR_SIZE])

```

Continued

```

14 {
15     for (int i=0; i < ARR_SIZE; i++)
16         outArr[i] = intArray[i];
17 }

```

Client of DirectToSOM C++ Class Array (array\starray.cpp):

```

1 #include <iostream.h>
2 #include <assert.h>
3 #include <somd.hh>
4 #include "check.h"
5 #include "array.hh"
6
7 int main(int argc, char *argv[])
8 {
9     __SOMEnv = SOM_CreateLocalEnvironment();
10
11     SOMD_Init(__SOMEnv);
12
13     // create remote object
14     Array *remoteObj =
15         (Array *)somdCreate(__SOMEnv, "Array", TRUE);
16     assert(! checkError(__SOMEnv, TRUE));
17
18     int arr[ARR_SIZE] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
19     remoteObj->setArray(arr);
20
21     int arrOut[ARR_SIZE];
22     for (int j=0; j < ARR_SIZE; j++)
23         arrOut[j] = 0;
24     remoteObj->getArray(arrOut);
25
26     cout << "arrOut: ";
27     for (int i=0; i < ARR_SIZE; i++)
28         cout << arrOut[i] << " ";
29     cout << endl;
30
31     remoteObj->somFree();
32
33     SOMD_Uninit(__SOMEnv);
34     SOM_DestroyLocalEnvironment(__SOMEnv);
35
36     return(0);
37 }

```

Passing SOMFOREIGN Types

With respect to allowable parameter types, DSOM supports remote method calls only if all parameters for a method are completely defined. From a DirectToSOM perspective, this means the parameter cannot contain a `void*` or a **SOMFOREIGN** type. DSOM does provide marshaling support for **SOMFOREIGN** types, but the class implementor must specify how that

marshaling should take place. There are three ways to handle marshaling of **SOMFOREIGN** types:

- ♦ opaque marshaling of binary data
- ♦ static marshaling function
- ♦ dynamic marshaling method

The simplest method is opaque marshaling, in which you specify that DSOM should simply copy the bytes from the client process to the server process. The other two methods involve defining a function or method to handle the marshaling. Note that dynamic marshaling support is currently available only for the C and C++ bindings, where marshaling information is generated with the binary.

In order to marshal foreign data, the following information is needed:

- ♦ the size in bytes of the data, specified with the IDL **length** modifier
- ♦ an **impctx** modifier that specifies:
 - ◊ the language of the foreign type
 - ◊ the storage class of the type, specified with the IDL **pointer** or **struct** modifier
 - ◊ plus one of the following:
 - opaque**, to indicate opaque data marshaling
 - static** with a function name to indicate for static marshaling
 - dynamic** with a method name to indicate dynamic marshaling

Whether **pointer** or **struct** is specified depends upon the underlying type. If the type is already an address (either a pointer, a string, or an object reference), then no extra indirection is required to pass the type. In this case, the parameter value is passed directly and **pointer** is specified. If the type requires an extra level of indirection to be passed, either a structure or an array where the address and not the value is passed, then **struct** is specified. The modifier that supplies the **SOMFOREIGN** marshaling information must appear before any use of that type.

The following example shows how to specify opaque marshaling for a **SOMFOREIGN** type. (See the SOMObjects documentation for details on defining functions for handling static or dynamic marshaling.) Recall from Chapter 7 that the compiler generates an **impctx** modifier for **SOMFOREIGN** types. Since there is currently no way to change that **impctx** modifier, you must use the **SOMIDLDecl** pragma to specify the complete IDL declaration that should be generated for a given type. The easiest way to do this is to generate the IDL first, then copy the result back into the **SOMIDLDecl** pragma string, modifying the **impctx** as appropriate. In this example, I specified `impctx = "C++, struct, opaque"`. (When the compiler no longer

generates **SOMFOREIGN** types as two pieces, you will not need the extra `__sfn` type definition.)

In order to ensure that the compiler passes the address of the structure and not its value, I specified a dummy destructor for the type `foreignType`. This is ANSI-compliant, in that a class with a destructor cannot be copied to the stack because the destructor can't be called, so it must be allocated as a temporary, whose address is passed. Otherwise, the structure could be copied to the stack and passed by value, which result in a run-time error because DSOM is expecting the address of the structure to be passed.

Definition of DirectToSOM C++ Class Foreign (foreign\foreign.hh):

```

1  #include <som.hh>
2
3  struct foreignType {
4      int i;
5      char c;
6      friend void bar();
7      // prevent pass-by-value (ANSI compliant)
8      ~foreignType() {};
9  };
10 #pragma SOMIDLDecl(foreignType, \
11     'typedef SOMFOREIGN sf0__;\n' \
12     'typedef sf0__ foreignType;\n' \
13     '*#pragma modifier sf0__ : * \
14     'impctx = \"C++,struct,opaque\", * \
15     'struct, \n\tsize = 8, length = 8, align = 4')
16
17 class Foreign : public SOMObject {
18     #pragma SOMClassName(*, "Foreign")
19     #pragma SOMIDLPass(*, "Implementation-End", \
20         "dllname = \"foreign.dll\";")
21     #pragma SOMIDLPass(*, "Implementation-Begin", \
22         "memory_management = corba;")
23     foreignType local;
24 public:
25     void set(foreignType);
26     foreignType get();
27 };

```

Implementation of DirectToSOM C++ Class Foreign (foreign\foreign.cpp):

```

1  #include <fstream.h>
2  #include "foreign.hh"
3
4  void Foreign::set(foreignType f)
5  {
6      cout << "Foreign::foo f.i: "
7          << f.i << " f.c: " << f.c << endl;
8      local = f;
9  }
10

```

```

11 foreignType Foreign::get()
12 {
13     return local;
14 }

```

Client of DirectToSOM C++ Class Foreign (foreign\tstfrgn.cpp):

```

1  #include <iostream.h>
2  #include <assert.h>
3  #include <somd.hh>
4  #include "check.h"
5  #include "foreign.hh"
6
7  int main(int argc, char *argv[])
8  {
9      __SOMEnv = SOM_CreateLocalEnvironment();
10
11      // Initialize DSOM run-time environment
12      SOMD_Init(__SOMEnv);
13      // Handle memory dealloc with delete/SOMFree
14      SOMD_NoORBfree();
15
16      // create remote object and assign proxy to remoteObj
17      Foreign *remoteObj =
18          (Foreign *)somdCreate(__SOMEnv, "Foreign", TRUE);
19      assert(! checkError(__SOMEnv, TRUE));
20
21      foreignType f = {10, 'c'};
22      cout << "f.i: " << f.i << ", f.c: " << f.c << endl;
23      remoteObj->set(f);
24      foreignType g = remoteObj->get();
25      cout << "f.i: " << f.i << ", f.c: " << f.c << endl;
26
27      // delete proxy and remote object
28      remoteObj->somFree();
29
30      SOMD_Uninit(__SOMEnv);
31      SOM_DestroyLocalEnvironment(__SOMEnv);
32
33      return(0);
34 }

```

Other Programming Considerations

Default Constructor

DSOM requires that all classes supply a default constructor. This is required when DSOM implicitly creates objects during parameter marshaling and any time an object is created through a factory specifying **somNew**. If you don't supply one, the VisualAge C++ compiler will generate one that causes a run-time error when called. So, it is recommended that you always supply a default constructor.

You can supply a nondefault constructor whose arguments all have default values. In this case, the compiler will generate a default constructor that calls the nondefault one, passing the default values for the parameters. In the next example, the compiler will supply a default constructor that calls `Hello(int)` passing the value 3.

Generated Default Constructor (*dftctr\dfctr.hh*):

```

1  #include <som.hh>
2
3  class DftCtr : public SOMObject {
4      #pragma SOMClassName(*, "DftCtr")
5      #pragma SOMIDLPass(*, "Implementation-End", \
6          "dllname = \"dfctr.dll\";")
7  public:
8      DftCtr(int = 3);
9      void sayHello();
10 };

```

Static Members

Static members are allocated and accessed locally in the current process. If a class has a static data member, each process (each client, plus any servers that implement the class) that references the class will have a separate copy of that static data. Multiple DSOM objects of the same class will not share static data unless those objects were created by the same process. For example, the two objects shown in Figure 8.4 would not share static data because they were created in different processes.

You can use static data members with DSOM, but only in the server process, and subject to the following restrictions:

- ♦ You can only access static data members indirectly through methods, that is, not directly through the member name. By using a method, the copy of the static data on the remote side will be accessed correctly.
- ♦ Static data members cannot be made into attributes, so to prevent inadvertent direct access from a client, the static member should be made private (or protected, but only if the derived class will be implemented by the same process as the base class). If you really need to access a static data member directly from the client as a data member, write a “wrapper” attribute with **noset**, **noget**, and **nodata** that operates on the static member remotely.
- ♦ All objects of the given class must be created by the same server process. If not, different copies of the static data member will be allocated for each server. This may be acceptable, but it is important to aware that this will take place.

In addition, static member functions are called directly through a pointer in the **ClassData** structure for the class, rather than through the SOM

method resolution mechanisms. The reason for using the **ClassData** structure at all, rather than calling directly from **DirectToSOM** C++, is to allow other languages to call the function without needing to know the name.

As with static data members, you can use static member functions with DSOM, but only from the server and only if the all objects of the class are created by the same server process. If the objects are created in different servers (as shown in Figure 8.4 on page 219), when a static member function is invoked against one of the objects the method contained in the invoking process will be called, regardless of where the target object was created. As with static data members, this may be acceptable. Note that even though the function is callable from the client side, the implementation may have supplied an updated version of that method, resulting in a mismatch between client and implementation. Static member functions should therefore also be made private to prevent inadvertent access from the client side.

Standard I/O

Many of the class implementation examples in this chapter print text to standard output, but you should not build a dependency upon standard I/O in your class implementation if that class will be used with DSOM because the implementation may be running in a server that doesn't have a display or keyboard.

Stub DLL

In all of the examples used in this chapter, the implementation DLL exists on the same machine, in the same directory, as the client program. If you want to keep the client and implementation separate, which occurs by definition when the client and server processes are on different machines, you may still need to provide a means through which the client can access the SOM class data structures. Even though the client is dealing with remote objects, if the client statically accesses the SOM class, the SOM class data structures must be available. (For details on the distinction between statically and dynamically using a class, refer to Chapter 5.)

Rather than having a complete class implementation on both the client and server machines, you can generate a DLL that contains only stubs for the class methods. This can be done by generating the C or C++ implementation templates for the class from the IDL description, and then compiling the implementation template into a DLL. The client can then link to this "stub" DLL in order to access the SOM class data structures.

Error Handling

DSOM uses the CORBA model for exception handling, in that exception information is returned to the caller through the **Environment** structure. Because each DSOM remote method call could potentially result in an error,

much more likely than with standard intraprocess interactions, you should check the **Environment** structure after every method invocation.

As discussed in Chapter 5, the DirectToSOM C++ compiler implicitly passes an **Environment** structure as the second parameter on every SOM method invocation. This structure is the compiler-declared global variable `__SOMEnv`. In your DSOM program, you will need to allocate and initialize this structure so that it can be passed to **SOMD_Init** for DSOM initialization, and on method invocations by the compiler. You can do this either dynamically through **SOM_CreateLocalEnvironment()**, or statically by assigning an automatic or static variable to `__SOMEnv` and calling **SOM_InitEnvironment()**. Then, after each method invocation, you should check the value of `__SOMEnv` to ensure that no errors occurred. If an error does occur, you must clear it using **somExceptionFree**, otherwise it may cascade and cause subsequent operations to fail. At the end of your program, you should deallocate this structure and clean up any memory that may have been allocated for it. I typically add a line such as:

```
assert(__SOMEnv->major == NO_EXCEPTION);
```

or

```
assert(checkError(__SOMEnv, TRUE));
```

after each method invocation or DSOM interaction when I start writing a program, just to catch any errors immediately. If you choose the former approach, later you should call an error-handling routine such as `checkError` (shown in the first example in this chapter) to perform better error handling and to deallocate any memory allocated for the exception.

By default, exception data from remote method calls is allocated differently from that of local method calls, and you must use **somdExceptionFree** and **somExceptionFree** respectively to deallocate the exception data. However, specifying **SOMD_NoORBfree** causes remote method exception storage to be allocated the standard way with **SOMMalloc**, so you can just use **somExceptionFree** to deallocate both types of exception data. Be aware that **somExceptionFree** performs only a shallow deallocation of the **Environment** structure. If the exception structure within the **Environment** structure contains embedded memory, you will need to step through the exception structure and explicitly deallocate this memory.

Customizing the Server

In many cases, the **somdsvr** program may be sufficient for handling server functionality. But there may be situations where you want to customize the server so that you can perform application-specific actions for each request. You must also provide your own server program if you want to replace the

SOM memory management routines in the server process, as discussed earlier. The server program must perform the following steps:

1. Initialize the DSOM environment.
2. Initialize the server implementation definition.
3. Initialize the SOM object adapter.
4. Indicate to the DSOM daemon, **somdd**, that it is ready to process requests.
5. Process requests as they are received.

The next example shows a simple server program. The program is invoked as follows:

```
myserver [impl_id | -a alias]
```

with either the server ID (generated when the server was registered in the implementation repository) or **-a** followed by the implementation alias name. When a server is invoked implicitly by the DSOM daemon, it is passed the server ID as the first parameter, so you must support at least the former in your server program, while supporting the latter makes it easier to start the server explicitly.

The DSOM environment is initialized at line 9. At lines 13 through 23, the server implementation definition is retrieved from the implementation repository and stored in the global variable **SOMD_ImplDefObject**. If the first parameter to the program is **-a**, the definition is retrieved at line 18 by invoking the method **find_impldef_by_alias** against the global object **SOMD_ImplRepObject**, passing the second program argument, which is assumed to be the implementation alias. If the first parameter to the program is not **-a**, the definition is retrieved by invoking the **find_impldef** method at line 23, passing the first argument to the program, which is assumed to be the server implementation ID.

At line 28, an object of type **SOMOA** is created and assigned to the global variable **SOMD_SOMOAObject**. This is the *object adapter* for the server process. The object adapter handles most of the server functionality, in that it is responsible for marshaling and demarshaling client requests, transforming them into operations on local objects. The **SOMOA** object is not created implicitly when DSOM is initialized because it is only needed in server processes.

Next, at line 33, the method **impl_is_ready** is invoked against the object adapter, passing the implementation definition. Among other things, this informs the DSOM daemon that the server program is ready to process requests. It also creates the object for the server, by default of type **SOMDServer**, which is assigned to the global variable **SOMD_ServerObject**. The object adapter uses the interface repository to determine the class of the server object. One of the purposes of the server object is to create and resolve object references. The object adapter invokes the **SOMDServer**

methods **somdSOMObjFromRef** and **somdRefFromSOMObj** against the **SOMD_ServerObject** to map between object references and the actual objects themselves.

At lines 37 through 45, the program loops, processing requests by executing the **execute_next_request** method against the object adapter object **SOMD_SOMOAObject**. The object adapter will demarshal the client request and invoke the appropriate method against the local object, and then marshal the result to return back to the client.

When the server program terminates, even abnormally, it should invoke **deactivate_impl** against the object adapter, as is done at line 48, to inform the DSOM daemon that the server is no longer active. This will allow a new server to be started and will prevent any further client requests against that server. Finally, at lines 50 through 52, the program cleans up any storage that it has allocated and terminates.

In order to specify that this program should be run as the server program for a given class, the program name must be specified with the server alias when registered in the implementation repository:

```
regimpl -A -i HelloServer -p myServer
```

This line indicates that the program **myServer.exe** should be executed as the server process for the **HelloServer** server process. This is shown in the makefile at line 28. Note that the program **myServer.exe** must be on the path for the **somdd** daemon, or **somdd** will not be able to find **myserver.exe** to execute it.

Figure 8.8 shows the object and process layout if the server program in this example were run with the program shown at the beginning of this chapter. All client requests through proxy objects are processed by the object adapter, which transforms them into method invocations against a local object through the **SOMD_ServerObject**.

Customized Server Program (svrpgm/myserver.cpp):

```

1  #include <iostream.h>
2  #include <assert.h>
3  #include <somd.hh>
4  #include "check.h"
5
6  int main(int argc, char *argv[])
7  {
8      __SOMEnv = SOM_CreateLocalEnvironment();
9      SOMD_Init(__SOMEnv);
10     checkError(__SOMEnv, TRUE);
11
12     cout << "retrieving implementation definition" << endl;
13     if (strcmp("-a", argv[1]) == 0)
14         // retrieve implementation definition from
15         // implementation repository by passing

```

Hello Application

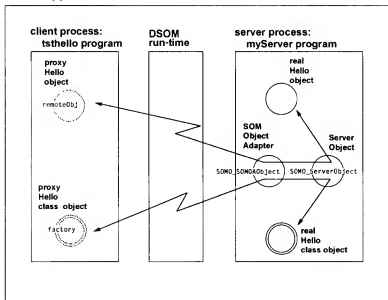


FIGURE 8.8 Server process model.

```

16         // implementation alias as key
17         SOMD_ImplDefObject =
18             SOMD_ImplRepObject->find_impldef_by_alias(argv[2]);
19     else
20         // retrieve implementation definition
21         // by passing implementation ID as key
22         SOMD_ImplDefObject =
23             SOMD_ImplRepObject->find_impldef(argv[1]);
24     checkError(__SOMEnv, TRUE);
25
26     cout << "creating Object Adapter" << endl;
27     // create an Object Adapter object
28     SOMD_SOMOAObject = new SOMOA;
29     checkError(__SOMEnv);
30
31     cout << "ready" << endl;
32     // indicate to DSOM daemon that ready to process requests
33     SOMD_SOMOAObject->impl_is_ready(SOMD_ImplDefObject);
34
35     // process requests
36     int count = 0;
37     for ( ;; ) {
38         int rc =
39             SOMD_SOMOAObject->execute_next_request(SOMD_NO_WAIT);
40         if (checkError(__SOMEnv, TRUE))
41             break;
42         if (! rc)

```

Continued

```

43         cout << "processed request number "
44             << ++count << endl;
45     }
46
47     // tell DSOM deamon that are deactivating
48     SOMD_SOMOAObject->deactivate_impl(SOMD_ImplDefObject);
49
50     delete SOMD_SOMOAObject;
51     SOMD_Uninit(__SOMEnv);
52     SOM_DestroyLocalEnvironment(__SOMEnv);
53
54     return(0);
55 }

```

Makefile (svrpgm/makefile):

```

1  all: hello.dll tsthello.exe myserver.exe somdimpl.dat
2
3  ICCOPTS = /I ..\include
4
5  hello.dll: hello.cpp hello.hh hello.i.c
6      icc $(ICCOPTS) /Ti+ /Ge- /Gd+ /B"/NOE" \
7      hello.cpp hello.i.c hello.def
8      implib hello.lib hello.dll
9
10 tsthello.exe: tsthello.cpp hello.hh
11     icc $(ICCOPTS) /Ti+ /B"/NOE" \
12     tsthello.cpp hello.lib ..\memoryl\sommeml.obj
13
14 myserver.exe: myserver.cpp
15     icc $(ICCOPTS) /Ti+ /B"/NOE" myserver.cpp \
16     ..\memoryl\sommeml.obj
17
18 hello.idl: hello.hh
19     icc $(ICCOPTS) hello.hh
20     sc -sir -u hello.idl
21
22 hello.i.c: hello.idl
23     sc -simod hello.idl
24     sc -sdef hello.idl
25
26 somdimpl.dat:
27     regimpl -D -i HelloServer
28     regimpl -A -i HelloServer -p myserver.exe
29     regimpl -a -i HelloServer -c Hello

```

Putting It All Together

As a final example, to illustrate many of the concepts discussed in a more realistic program, the message queue program has been updated to support distributed access. Different programs can call the client `tstmq` program simultaneously and share the message queues at the server.

The class definitions and implementations are much the same as before. The basic difference is that the `MessageQueueManager` and each `MessageQueue` object are now implemented as remote objects, for which a proxy is created. The proxy ID is stored locally in a file, so on each request for a message queue, the local file is checked first. If the proxy is found locally, it is used, otherwise a new remote object is created. I added a wrapper class, `LocalMessageQueueManager`, that will handle the remote object creation aspects. In addition, the use of a remote object requires changes to some methods to handle memory management and parameter passing correctly.

The `mqueue.hh` header has been updated to allow the `MessageQueue` class to be used in a DSOM environment. At lines 12 through 16, the **SOMClassName** and **SOMIDLPass** pragmas are specified to provide class name and memory management information. At lines 27 and 28, the data member `name` is made into a **readonly** attribute, with **object_owns_result** memory management for the `_get` method. At line 38, the IDL signature for the `Receive` method is specified with **out** for the string parameter (the default is **inout**).

The `MessageQueueManager` class in `mqmgr.hh` has also been updated for DSOM enablement. First the file extension has changed from `.h` to `.hh`, indicating that the class is now a SOM class. Lines 12 through 16 specify pragmas to provide the class name and memory management information, as for the `MessageQueue` class.

The file `lmqmgr.h` contains the definition of the class `LocalMessageQueueManager`, which is used to wrap the remote object management for the program. An instance of this class is declared in the client program, rather than an instance of the `MessageQueueManager` class in the earlier versions of this application. This class has a static member `mqmgr`, which stores the single `MessageQueueManager` proxy used by the application. The two private methods `ReadProxyFromFile` and `WriteProxyToFile` manage the conversion of a proxy to a string ID and vice-versa. The constructor performs DSOM initialization for the application. Like the `MessageQueueManager` class, this class also supports two versions of the `GetMessageQueue` method, which are invoked by the client to retrieve a message queue. These methods will handle checking for a local proxy ID and creating a remote object if one doesn't exist.

The implementation for the message queue class in `mqueue.cpp` is almost identical to the non-DSOM version. The only difference is in the **Receive** method implementation, lines 58 through 78. Because the class uses CORBA memory management, the **out** parameter will be deallocated at the server side on return to the client. Therefore, the method must allocate storage for the return value that can safely be deallocated by DSOM, rather than simply returning the address of the underlying data.

The implementation of the `MessageQueueManager` class is identical to the previous versions. The `GetMessageQueue` methods do not allocate special storage for DSOM to deallocate on return from the method. This is not

necessary because the `MessageQueueManager` and the `MessageQueue` class are implemented in the same server process, so the `MessageQueue` objects being returned are local to the `MessageQueueManager` process. DSOM will implicitly create a corresponding **SOMDObject** for the local object being returned, which will be marshaled and released on return from the method instead of the local object itself. If, however, the `MessageQueueManager` and the `MessageQueue` classes were implemented in different processes, then the objects being returned would be proxies, in which case the methods would need to make a copy of the proxy, using the **SOMDObject::duplicate** method, and return that copy for DSOM to deallocate.

The `lmqmgr.cpp` file, which contains the implementation of the `LocalMessageQueue` class, contains the bulk of the DSOM enablement. The constructor, lines 13 through 54, initializes the DSOM environment and looks for the message queue manager in the local file `server.lcl`. If the proxy is retrieved and it represents a valid object, the constructor terminates. If the proxy is retrieved and it does not represent a valid remote object, all other proxy files are erased. (Each message queue proxy is stored in an individual file, so erasing the other proxy files ensures that no other invalid proxy objects will be retrieved during program execution). Then a remote `MessageQueueServer` object is created at line 49. The ID is written to the proxy file, and the constructor terminates.

The `LocalMessageQueueManager::GetMessageQueue(int)` method, lines 64 through 68, simply forwards the request to the remote message queue manager. The `LocalMessageQueueManager::GetMessageQueue(char)` method, lines 75 through 96, searches for the requested message queue locally and returns the converted proxy if found. Otherwise, it creates a new remote message queue and saves it locally, using the message queue name as the file name, with an extension of `.pxy`.

This is certainly not the most efficient way to write such an application. For example, it would be much more efficient to keep the proxy pointers for each message queue in memory locally, rather than going to the file each time. But I wanted to keep the example as simple as possible in order to concentrate on the DSOM aspects.

The client program in file `tstmq.cpp` is almost identical to previous versions. The main difference is that `mqlist` is declared as type `LocalMessageQueueManager` at line 17, whereas it was of type `MessageQueueManager` in previous versions. In addition, the returned DSOM-allocated memory is deallocated at the end of each loop in lines 77 through 83.

**Definition of DirectToSOM C++ Class `MessageQueue`
(`mqqueue/mqueue.hh`):**

```
1 #ifndef MQQUEUE_H
2 #define MQQUEUE_H
3
4 #include <som.h>
```



```

5
6 #define SUCCESS 0
7 #define FAIL 1
8 #define MAX_QUEUE_NAME_LEN 20
9 #define MAX_MESSAGE_LEN 256
10
11 class MessageQueue : public SOMObject {
12     #pragma SOMClassName(*, "MessageQueue")
13     #pragma SOMIDLPass(*, "Implementation-End", \
14         "dllname = \"mqqueue.dll\";")
15     #pragma SOMIDLPass(*, "Implementation-Begin", \
16         "memory_management = corba;")
17     struct Mqueue {
18         Mqueue *next;
19         char* message;
20         Mqueue(char *);
21         ~Mqueue();
22     };
23     Mqueue *mq, *last;
24     int count;
25 public:
26     char *name;
27     #pragma SOMAttribute(name, readonly)
28     #pragma SOMIDLPass(*, "Implementation-End", \
29         "_get_name : object_owns_result;")
30     MessageQueue();
31     MessageQueue(char *);
32     ~MessageQueue();
33     virtual int Count();
34     virtual void Clear();
35     virtual int Send(char *);
36     virtual int Receive(char **);
37     #pragma SOMMethodName(Receive, "Receive")
38     #pragma SOMIDLDecl(Receive, \
39         "long Receive(out string arg1)")
40     virtual void Dump();
41
42     #pragma SOMReleaseOrder( \
43         /* 1 */ MessageQueue(char*), \
44         /* 2 */ Clear(), \
45         /* 3 */ Send(char*), \
46         /* 4 */ Receive(char**), \
47         /* 5 */ Dump(), \
48         /* 6 */ Count(), \
49         /* 6 */ name)
50 };
51
52 #endif

```

**Definition of DirectToSOM C++ Class MessageQueueManager
(mqqueue/mqmgr.hh):**

```

1 #ifndef MQSERVER_H
2 #define MQSERVER_H

```

Continued

```

3
4 #include <som.hh>
5 #include "mqueue.hh"
6
7 #define MAX_QUEUES 20
8
9 class MessageQueueManager : public SOMObject {
10     MessageQueue *mqueues[MAX_QUEUES];
11     public:
12     #pragma SOMClassName(*, "MessageQueueManager")
13     #pragma SOMIDLPass(*, "Implementation-End", \
14         "dllname = \"mqmgr.dll\";")
15     #pragma SOMIDLPass(*, "Implementation-Begin", \
16         "memory_management = corba;")
17     MessageQueueManager();
18     ~MessageQueueManager();
19     virtual MessageQueue *GetMessageQueue(char *);
20     virtual MessageQueue *GetMessageQueue(int);
21 };
22
23 #endif

```

**Definition of Native C++ Class LocalMessageQueueManager
(mqueue/lmqmgr.h):**

```

1 #ifndef LMQSERVER_H
2 #define LMQSERVER_H
3
4 #include "mqmgr.hh"
5
6 class LocalMessageQueueManager {
7     MessageQueueManager *mqmgr;
8
9     SOMObject* ReadProxyFromFile(char *fname);
10    void WriteProxyToFile(char *fname, SOMObject *remoteObj);
11    public:
12    LocalMessageQueueManager();
13    ~LocalMessageQueueManager();
14    virtual MessageQueue *GetMessageQueue(char *);
15    virtual MessageQueue *GetMessageQueue(int);
16 };
17
18 #endif

```

**Implementation of DirectToSOM C++ Class MessageQueue
(mqueue/mqueue.cpp):**

```

1 #include <iostream.h>
2 #include <assert.h>
3 #include "mqueue.hh"
4
5 MessageQueue::Mqueue::Mqueue(char *elemMessage)
6 {

```

```

7     next = NULL;
8     message = new char[strlen(elemMessage) + 1];
9     assert(message != NULL);
10    strcpy(message, elemMessage);
11 }
12
13 MessageQueue::Mqueue::~Mqueue()
14 {
15     delete message;
16     if (next)
17         delete next;
18 }
19
20 MessageQueue::MessageQueue()
21 {
22     name = NULL;
23     last = mq = NULL;
24     count = 0;
25 }
26
27 MessageQueue::MessageQueue(char *qname)
28 {
29     last = mq = NULL;
30     count = 0;
31     name = new char[strlen(qname) + 1];
32     assert(name != NULL);
33     strcpy(name, qname);
34 }
35
36 MessageQueue::~MessageQueue()
37 {
38     Clear();
39     if (name)
40         delete name;
41 }
42
43 int MessageQueue::Send(char *message)
44 {
45     Mqueue *elem;
46     if (! (elem = new Mqueue(message)))
47         return FAIL;
48     if (mq == NULL) {
49         mq = last = elem;
50     } else {
51         last->next = elem;
52         last = elem;
53     }
54     ++count;
55     return SUCCESS;
56 }
57
58 int MessageQueue::Receive(char **bufp)
59 {
60     if (!mq) {
61         // can return NULL with DSOM 3.0

```

Continued

```

62         *bufp = NULL;
63         return FAIL;
64     }
65     Mqueue *elem = mq;
66     mq = mq->next;
67     --count;
68     if (last == elem)
69         last = NULL;
70     // DSOM will deallocate this if caller-owned
71     *bufp = new char(strlen(elem->message) + 1);
72     assert(*bufp);
73     strcpy(*bufp, elem->message);
74     // so don't delete entire chain
75     elem->next = NULL;
76     delete elem;
77     return SUCCESS;
78 }
79
80 int MessageQueue::Count()
81 {
82     return count;
83 }
84
85 void MessageQueue::Dump()
86 {
87     int i = 1;
88     cout << "Dumping queue " << name << endl;
89     for (Mqueue *cur = mq; cur != NULL;
90          cur = cur->next, i++)
91         cout << '\t' << i << ": "
92              << cur->message << endl;
93 }
94
95 void MessageQueue::Clear()
96 {
97     if (mq != NULL) {
98         delete mq;
99         mq = last = NULL;
100     }
101     count = 0;
102 }

```

***Implementation of DirectToSOM C++ Class MessageQueueManager
(mqmgr.cpp):***

```

1  #include <iostream.h>
2  #include "assert.h"
3  #include "mqmgr.hh"
4
5  #define MAX_QUEUES 20
6
7  MessageQueueManager::MessageQueueManager()
8  {
9      for (int i=0; i<MAX_QUEUES; i++)

```

```

10         mqueues[i] = NULL;
11     }
12
13     MessageQueueManager::~MessageQueueManager()
14     {
15         for (int i=0; i<MAX_QUEUES; i++)
16             if (mqueues[i])
17                 delete mqueues[i];
18     }
19
20     MessageQueue *MessageQueueManager::
21     GetMessageQueue(char *name)
22     {
23         for (int i=0; i<MAX_QUEUES; i++)
24             if (mqueues[i] &&
25                 mqueues[i]->name &&
26                 strcmp(mqueues[i]->name, name) == 0)
27                 return mqueues[i];
28         for (i=0; i<MAX_QUEUES && mqueues[i]; i++)
29             ;
30         if (i == MAX_QUEUES)
31             return NULL;
32         if (! (mqueues[i] = new MessageQueue(name))) {
33             return NULL;
34         }
35         return mqueues[i];
36     }
37
38     MessageQueue *MessageQueueManager::
39     GetMessageQueue(int qnum)
40     {
41         if (qnum < MAX_QUEUES && mqueues[qnum])
42             return mqueues[qnum];
43         else
44             return NULL;
45     }

```

**Implementation of Native C++ Class LocalMessageQueueManager
(mqueue/lmqmgr.cpp):**

```

1  #include <fstream.h>
2  #include <iostream.h>
3  #include <stdio.h>
4  #include <assert.h>
5
6  #include "lmqmgr.h"
7  #include "check.h"
8
9  #pragma SOMNODataDirect(on)
10 #include "mqmgr.hh"
11 #pragma SOMNODataDirect(off)
12
13 LocalMessageQueueManager::LocalMessageQueueManager()
14 {

```

Continued

```

15     // initialize DSOM
16     static Environment env;
17     SOM_InitEnvironment(__SOMEnv = &env);
18     SOMD_Init(__SOMEnv);
19     // Handle memory dealloc with delete/SOMFree
20     SOMD_NoORBFree();
21
22     char *fname = "server.lcl";
23     // look for the server locally first
24     mqmgr = (MessageQueueManager *)
25         ReadProxyFromFile(fname);
26
27     if (mqmgr && ! isValidRemoteObject(mqmgr)) {
28         // system-dependent file operation
29         system("erase *.lcl");
30         mqmgr = NULL;
31     }
32
33     if (! mqmgr ) {
34         // Get Naming Service factory service
35         ExtendedNaming::ExtendedNamingContext *enc =
36             (ExtendedNaming::ExtendedNamingContext *)
37             SOMD_ORBObject->resolve_initial_references(
38                 "FactoryService");
39         assert(enc && ! checkError(__SOMEnv, TRUE));
40
41         // Find factory (class object)
42         // for class MessageQueueManager
43         SOMClass *factory = (SOMClass *)
44             enc->find_any("class == 'MessageQueueManager'"
45                 " and alias == 'MQServer'", 0);
46         assert(factory && ! checkError(__SOMEnv, TRUE));
47
48         // Create a remote object through the factory
49         mqmgr = (MessageQueueManager *)factory->somNew();
50         assert(mqmgr && ! checkError(__SOMEnv, TRUE));
51
52         WriteProxyToFile(fname, mqmgr);
53     }
54 }
55
56 LocalMessageQueueManager::~LocalMessageQueueManager()
57 {
58     ({SOMDObject *}mqmgr)->release();
59 }
60
61
62 // returns the proxy for the remote
63 // MessageQueue by queue number
64 MessageQueue* LocalMessageQueueManager::
65     GetMessageQueue(int qnum)
66 {
67     return mqmgr->GetMessageQueue(qnum);
68 }

```

```

69
70
71 // returns the proxy for the remote MessageQueue
72 // by queue name queues are stored locally by name,
73 // so a local search is performed
74 // first for the queue before checking the server.
75 MessageQueue* LocalMessageQueueManager::
76     GetMessageQueue(char *qName)
77 {
78     MessageQueue *remoteObj = NULL;
79
80     char *fname = new char[strlen(qName) + 5];
81     assert(fname != NULL);
82     sprintf(fname, "%s.lcl", qName);
83
84     // look for the message queue locally first
85     remoteObj =
86         (MessageQueue *)ReadProxyFromFile(fname);
87
88     if (! remoteObj) {
89         // file/object doesn't exist, so get object
90         // from server, and save the id for the
91         // object in the file
92         remoteObj = mgr->GetMessageQueue(qName);
93         WriteProxyToFile(fname, remoteObj);
94     }
95     return remoteObj;
96 }
97
98
99 // Reads the id from the given file and
100 // returns corresponding proxy
101 SOMObject* LocalMessageQueueManager::
102     ReadProxyFromFile(char *fname)
103 {
104     ifstream proxyFile(fname);
105     if (! proxyFile)
106         return NULL;
107     char proxyId[256];
108     proxyFile >> proxyId;
109     proxyFile.close();
110     SOMObject *remoteObj = (SOMObject *)
111         SOMD_ORBObject->string_to_object(proxyId);
112     return remoteObj;
113 }
114
115
116 // Writes the id for a proxy to the given file
117 void LocalMessageQueueManager::
118     WriteProxyToFile(char *fname, SOMObject *remoteObj)
119 {
120     ofstream proxyFile(fname);
121     char *id;
122

```

Continued


```

45         }
46         break;
47     case 'l': case 'L':
48         int i;
49         for (i=0; i < MAX_QUEUES; i++) {
50             if ( (mq=mqlist.GetMessageQueue(i)) != NULL)
51                 cout << "Name: " << mq->name << " count: "
52                     << mq->Count() << endl;
53         }
54         break;
55     case 'd': case 'D':
56         cout << "Enter queue name: ";
57         cin >> qname;
58         if ( (mq=mqlist.GetMessageQueue(qname)) != NULL)
59             mq->Dump();
60         break;
61     case 'n': case 'N':
62         cout << "Enter queue name: ";
63         cin >> qname;
64         if ( (mq=mqlist.GetMessageQueue(qname)) != NULL)
65             cout << "Size of queue " << qname
66                 << ": " << mq->Count() << endl;
67         break;
68     case 'c': case 'C':
69         cout << "Enter queue name: ";
70         cin >> qname;
71         if ( (mq=mqlist.GetMessageQueue(qname)) != NULL)
72             mq->Clear();
73         break;
74     case 'q': case 'Q':
75         return 0;
76
77     if (msgp)
78         // delete DSOM-allocated memory
79         delete msgp;
80     if (mq)
81         // deallocate proxy
82         {(SOMDObject *)mq}->release();
83     }
84 }
85 return(0);
86 }

```

Makefile (mqueue/makefile):

```

1  all: mqueue.dll mgmgr.dll tstmq.exe somdimpl.dat
2
3  ICCOPTS = /I ..\include
4
5  mqueue.dll: mqueue.hh mqueue.cpp mqueuei.c
6      icc $(ICCOPTS) /Ti+ /Ge- /B"/NOE" mqueue.cpp \
7      mqueuei.c mqueue.def ..\memory1\sommem1.obj
8      implib mqueue.lib mqueue.dll
9

```

Continued

```

10  mqmgr.dll: mqmgr.hh mqmgr.cpp mqmgr.i.c
11  icc $(ICCOPTS) /Ti+ /Ge- /B*/NOE" mqmgr.cpp mqmgr.i.c \
12  mqmgr.def ..\memory1\sommem1.obj mqueue.lib
13  implib mqmgr.lib mqmgr.dll
14
15  tstmq.exe: mqueue.hh mqmgr.hh tstmq.cpp lmqmgr.h lmqmgr.cpp
16  icc $(ICCOPTS) /Ti+ /B*/NOE" tstmq.cpp lmqmgr.cpp \
17  ..\memory1\sommem1.obj mqueue.lib mqmgr.lib
18
19  mqueue.idl: mqueue.hh
20  icc $(ICCOPTS) mqueue.hh
21  sc -sir -u mqueue.idl
22
23  mqmgr.idl: mqmgr.hh
24  icc $(ICCOPTS) mqmgr.hh
25  sc -sir -u mqmgr.idl
26
27  mqueuei.c: mqueue.idl
28  sc -simod mqueue.idl
29  sc -sdef mqueue.idl
30
31  mqmgr.i.c: mqmgr.idl
32  sc -simod mqmgr.idl
33  sc -sdef mqmgr.idl
34
35  somdimpl.dat:
36  regimpl -D -i MQServer
37  regimpl -A -i MQServer
38  regimpl -a -i MQServer -c MessageQueueManager

```

Common DSOM Problems

Debugging the Server

In the examples so far, the server program has always been started automatically when needed, but you can also explicitly start the server program from the command line. This is useful for debugging the server side, as you can start the server through the debugger. If you are using the generic SOM server, the program name is **somdsvr**. The syntax for starting this program is:

```
somdsvr [impl_id | -a alias]
```

where *impl_id* is the implementation ID for the server and *alias* is the server alias name. The alias name is that the server, which you gave with the **regimpl** utility to register the server in the implementation repository. The implementation ID is available from the **regimpl** utility or through the implementation repository APIs.

For example, to explicitly start the **somdsvr** program for the message queue example, the following would be used:

```
somdsvr -a MQServer
```

To start the server through the debugger in OS/2 or Windows:

```
ipmd somdsvr -a MQServer
```

You can then create a load occurrence breakpoint in **ipmd** corresponding to the loading of the SOM class DLL so that you can debug it. (Don't forget to create both the object and the DLL with debug mode turned on.) Another option is to pipe the output of the server to a file:

```
somdsvr -a MQServer > somdsvr.out
```

DSOM Error Messages

There are many messages that may show up in the SOM error log when you are getting started with DSOM. The SOM error log file is given by the **SOMErrorLogFile** setting in the **somenv.ini** configuration file. By default, the file is **somerror.log** in the directory given by the **SOMDDIR** setting in **somenv.ini**. In addition, SOM error log messages are written to standard output if the **SOMErrorLogDisplayMsgs** setting is set to yes, which is the default. (Note: In DSOM 2.x, DSOM errors are written to the file specified by the environment variable **SOMDMESSAGELOG** if the environment variable **SOMDDEBUG** is set to 1.) The errors that I encountered initially, and most frequently, are discussed next.

30016: SOMDERROR_BadDescriptor

This indicates that DSOM could not find the appropriate information about the class in the interface repository. Either the class has not been compiled into the interface repository at all, or the interface repository class information does not match the current class definition. Use the **IRDUMP** utility, specifying the class name, to examine the interface repository contents.

30046: SOMDERROR_ImplNotFound

This occurs in DSOM 2.x when the DSOM object manager is not able to find the implementation for the class in the implementation repository. This is likely caused by the class server not being registered in the implementation repository, or being registered incorrectly. See the explanation for **SOMDERROR_ClassNotFound** next.

30047: SOMDERROR_ClassNotFound

This results when DSOM cannot find information about the class. There are several possible causes:

1. *The class server is not registered in the implementation repository, or is registered incorrectly:* The class name specified in the client program (for example, with the **somdCreate** method) must match exactly the class name specified when the implementation is registered with the **regimpl** or **pregimpl** utilities. To determine if the implementation is registered correctly, run the **regimpl** or **pregimpl** utilities with no options and select option 10 to list all the classes supported by each implementation. Check that you have registered a server to support the class and that the class names match exactly (note that the name checking is case-sensitive).
2. *The class has not been compiled into the interface repository correctly:* Make sure the class name you have specified when you registered the server matches the class compiled into the interface repository. The name checking between the interface repository and the implementation repository is not case-sensitive. You can use the **irdump** utility, specifying the class name as the first argument, to dump the contents of the interface repository. Note that **irdump** is case-sensitive.
3. *The DLL for the class cannot be found:* Check that the class DLL is on a path that can be found by the client, the DSOM daemon **somdd** (if the server process will be started automatically by **somdd**), and the server program (if you are starting the server process explicitly).

If the DLL is already loaded, DSOM just uses the loaded version. Otherwise, the DLL must be on the library path for DSOM to be able to load it. If the application is a Workstation application that statically loads the implementation DLL prior to requesting that DSOM load the DLL, the DLL will always be loaded before DSOM needs it. If, however, you start the server program explicitly prior to running the client, the DLL may not be loaded yet, and it must be on the library path in this situation. This has caught me a couple of times, where I had a program that ran perfectly well if the server process was started automatically by DSOM, but wouldn't work if I tried to start the server process explicitly. There are other situations where this can cause problems, too, so it's important to be aware of when the DLL is being loaded and who is loading it.

4. *The DLL is not being initialized properly by the **SOMInitModule** initialization function:* Verify that the DLL contains a **SOMInitModule** initialization function. This function is generated by the **imod** init/term function emitter, so make sure that you have generated the init/term function and have linked it into the DLL. For DSOM 2.x, you must explicitly supply the **SOMInitModule** function yourself.

30109: SOMDERROR_SOMDDNotRunning**30088: SOMDERROR_NamingNotActive**

These messages indicate that the DSOM daemon and the naming service are not active. They also seem to occur when DSOM encounters an unexpected problem. Two situations that I have encountered are attempting to use an invalid proxy object where the remote object is no longer valid because the remote server has terminated, and not clearing an exception before attempting another DSOM operation.

If you get an error that doesn't make sense in the current context—for example, if **somdd** is active when you get a **30109**—it is likely that some kind of unexpected problem occurred. Ensure that all object references are valid and that you have cleared any exceptions. I find this behavior much like a C compiler—a strange error that is the result of an earlier error can be flagged by the compiler later in a compilation unit. As with compiler syntax errors, the rule of thumb is to fix the problems you understand first, and often the confusing ones go away.

Parameter Data Not Returned

Whenever you are expecting data to be returned from the remote side as a parameter (as opposed to a function return value), the IDL description must define that parameter as either **out** or **inout**. You can verify this by checking the IDL file or by dumping the IR with the **irdump** command. This error is most likely to occur when you are passing a `char *` type and are expecting the remote side to update the underlying string. Recall that a `char *` maps to a **string** in IDL, which will map to an **in** parameter. In order to have a **string** returned as a parameter, you must define the parameter as `char **`, which will map to **inout**. You can use the **SOMIDLDecl** pragma, as shown for the `MessageQueue::Receive` method in the message queue example, to force the parameter to be **out**.

String or Object Value Lost

Whenever you are attempting to copy or assign a string or object to a data member, be sure that you understand the memory management being used. For example, if you pass a string to a remote object using **caller-owned** memory management, DSOM will allocate storage on the server side to hold the incoming string; however, it will deallocate this storage on return from the method invocation. If you simply save the address of the input string in the remote object, that address and its underlying storage will become invalid upon return from the method invocation.

Problems Starting Custom Server Program

Confirm that the program is registered correctly in the implementation definition for the server alias. Next make sure that the program is on the path used by the SOM daemon (**somdd**). Try explicitly starting the server program yourself, passing the server ID as the first parameter. The server ID can be obtained from the implementation repository using the **regimpl** or **pregimpl** utilities; or, if you have written your server to support the **-a** option, you can pass the server alias ID instead of the implementation ID.

Exception in SOM Run-Time or Other SOM-enabled Class

There are many possible causes of an exception when you are invoking a SOM method, but one likely is that the **Environment** parameter has not been allocated. Later versions of the compiler will explicitly assign storage to **__SOMEnv** at program startup, but earlier versions of the compiler did not. Make sure that **__SOMEnv** is allocated, otherwise an exception will occur if the target method encounters an error situation and attempts to update the **Environment** parameter. While this is not specific to DSOM, it is more common for DSOM applications to update the **Environment** parameter.

Exception when Deallocating Storage

If an exception occurs during a **delete** or **somFree** operation, a likely cause is that the memory was allocated through a different memory manager from the deallocation routine. If the memory was allocated with the C++ memory allocator, you must use the C++ deallocation routines, and the same for the SOM memory allocation routines. In order to avoid problems with memory allocator mismatches, you should either map the C++ routines to the SOM memory allocation routines or vice-versa, as discussed earlier in this chapter.

Checklist

The following is a quick checklist of common problems to look for if you are having difficulty getting your DSOM program running:

1. Check that the implementation repository has been correctly updated for the class.
2. Verify that the IR path is correct and identical for both the client and the server; and that the IR contents are consistent with the current class definition.
3. Ensure that the class DLLs can be found by the client and the server.

4. Make sure that all memory is being allocated through the same allocator by mapping the SOM routines to the C/C++ library routines or vice-versa. An error occurring on a memory deallocation operation is frequently caused by a memory allocator mismatch.
5. For all pointer and object parameters, make sure you understand the memory management that will take place in order to avoid invalid deallocation attempts by DSOM (for example, of static storage), memory leaks, and dangling references.
6. Ascertain that the `__SOMEnv` variable is allocated, and that you are checking for, and clearing, any exceptions.
7. Make sure that all public instance data is made into attributes. Specify **SOMNoDataDirect** for the class in the client code. If you will be creating objects for a derived class and its base class in different servers, make all protected instance data into attributes. Specify **SOMNoDataDirect** for the base class in the derived class code. If you will be creating objects for a class in different servers, make all private data into attributes. Specify **SOMNoDataDirect** in the class header file.
8. Ensure that you have defined the attribute routines for all pointer and object attributes so that memory leaks and dangling pointers don't occur.

Refer to the SOMObjects DSOM documentation for a checklist of DSOM configuration options.

DSOM 2.x

This section describes the programming considerations for using DSOM 2.x, if you do not yet have SOMObject 3.0 installed on your system. The main differences between DSOM 2.x and DSOM 3.0 are in how objects are created and DSOM memory management. DSOM 3.0 also lifts some of the restrictions in DSOM 2.x, such as the parameter types supported. The configuration for DSOM 2.x is the same as for DSOM 3.0, except that the Naming Service is not available for DSOM 2.x and thus the `som_cfg` command is not required, or available.

Creating Remote Objects

In DSOM 2.x, objects are created through *server* objects, rather than factory objects as in DSOM 3.0. You can request that an object be created through any server that supports the class, similar in concept to creating a DSOM 3.0 object through any factory that supports the class. This is achieved through the DSOM Object Manager method `somdNewObject`, as shown in the next example.

The main difference between DSOM 3.0 and DSOM 2.x in this case is at line 17, where the **somdNewObject** method is invoked against the **SOMD_ObjectMgr** object to create a remote object and its proxy. Two parameters can be passed to the **somdNewObject** method, the first being the class name for the remote object, in this case **Hello**, and the second being “hints” on how to find a server for that class. **SOMD_ObjectMgr** is an instance of the DSOM Object Manager, which is used to create, destroy, find, and identify remote objects in DSOM 2.x. The SOM Object Manager is created as part of DSOM initialization when **SOMD_Init** is invoked.

The other difference is at line 24, where the **somDestroyObject** method is invoked against the DSOM Object Manager to destroy both the remote and the proxy object. With DSOM 3.0, **somFree** is used, which in DSOM 2.x deallocates only the target object.

Creating a Remote Object with DSOM 2.x (dsom21\hello\tsthello.cpp):

```

1  #include <iostream.h>
2  #include <assert.h>
3  #include <somd.hh>
4  #include "hello.hh"
5
6  int main(int argc, char *argv[])
7  {
8      __SOMEnv = SOM_CreateLocalEnvironment();
9
10     Hello localObj;
11     localObj.sayHello();
12
13     SOMD_Init(__SOMEnv);
14     assert(__SOMEnv->_major == NO_EXCEPTION);
15
16     // create remote object
17     Hello *remoteObj = (Hello *)
18         SOMD_ObjectMgr->somdNewObject("Hello", "");
19     assert(__SOMEnv->_major == NO_EXCEPTION);
20
21     // Use proxy if created to issue message
22     remoteObj->sayHello();
23
24     SOMD_ObjectMgr->somdDestroyObject(remoteObj);
25
26     SOMD_Uninit(__SOMEnv);
27     SOM_DestroyLocalEnvironment(__SOMEnv);
28
29     return(0);
30 }
```

In a similar fashion to requesting a specific factory in DSOM 3.0, you can request that an object be managed through a specific server in DSOM 2.x. To do this, you can ask the DSOM Object Manager to locate a particular

server implementation, either by name, by ID, or by the class that it supports. You can request a specific server through one of the following four DSOM Object Manager methods:

```
SOMDServer* somdFindServerByName(string servername)
SOMDServer* somdFindServer(Implid serverid)
SOMDServer* somdFindAnyServerByClass(Identifier objclass)
_IDL_SEQUENCE_SOMDServer somdFindServersByClass(Identifier objclass)
```

somdFindServerByName uses the server's implementation alias name to locate the appropriate server. **somdFindServer** uses the server's implementation ID, a unique identifier string generated when the implementation is registered, to locate the server. **somdFindAnyServerByClass** finds any server that implements the given class, and **somdFindServersByClass** returns a list of all the servers that implement the particular class. These methods allow an application more flexibility in determining the location and implementation for remote objects.

As discussed earlier in *Customizing the Server*, each server process always contains a server object, which provides an interface for managing objects created in the remote process. When a client requests a specific server implementation it is returned a proxy to the server object in the corresponding remote process. This proxy is just like the proxy for a standard object, except that it represents a server object. By default, the server object for a class will be of type **SOMDServer**. It supports interfaces such as **somdCreateObj** and **somdDeleteObj**, which you can use to create and destroy objects. Customized server classes can be defined by inheriting from **SOMDServer**, but this approach is not recommended for migration to DSOM 3.0.

The next example shows how to use the **somdFindServerByName** method to find a server object with the alias name `HelloServer`. At line 18, the variable `serverProxy` will be assigned the proxy object for the remote server, if the method **somdFindServerByName** executes successfully. Then, at line 23, **somdCreateObj** is invoked against the server proxy object, supplying it with the class of the object to create. At line 31, the remote object and its proxy are deleted through the server by invoking the **somdDeleteObj** method. Finally, at line 35, the proxy for the server object is deleted through the **somdReleaseObject**, which deletes the proxy object but keeps the remote server object active (in DSOM 3.0, the use of **release** is recommended instead of **somdReleaseObject**).

Creating a Remote Object through a Server Object ***(dsom21\findsvr\tsthello.cpp):***

```
1 #include <iostream.h>
2 #include <assert.h>
3 #include <somd.hh>
```

Continued

```

4  #include "check.h"
5  #include "hello.hh"
6
7  int main(int argc, char *argv[])
8  {
9      __SOMEnv = SOM_CreateLocalEnvironment();
10
11     Hello localObj;
12     localObj.sayHello();
13
14     SOMD_Init(__SOMEnv);
15     assert(! checkError(__SOMEnv, TRUE));
16
17     // Find server with alias name HelloServer
18     SOMDServer *serverProxy =
19         SOMD_ObjectMgr->somdFindServerByName("HelloServer");
20     assert(! checkError(__SOMEnv, TRUE));
21
22     // Create an object through the server proxy
23     Hello *remoteObj = (Hello *)
24         serverProxy->somdCreateObj("Hello", "");
25     assert(! checkError(__SOMEnv, TRUE));
26
27     // Use proxy if created to issue message
28     remoteObj->sayHello();
29
30     // Delete the proxy and remote object
31     serverProxy->somdDeleteObj(remoteObj);
32
33     // Delete the local server proxy,
34     // but not the remote server object
35     SOMD_ObjectMgr->somdReleaseObject(serverProxy);
36
37     SOMD_Uninit(__SOMEnv);
38     SOM_DestroyLocalEnvironment(__SOMEnv);
39
40     return(0);
41 }

```

Figure 8.9 shows the object and process layout for the program in above. Note that the client contains a proxy for both the `Hello` object and the server object. The remote side always contains a server object, but the local side only contains a proxy for that server if it specifically requested it.

Memory Management

In DSOM 2.x, the default memory management is **caller-owned** for **in** and **inout** parameters, **dual-owned** for **out** parameters and return results, and **suppress_inout_free** for **inout** parameters. For ease of migration to DSOM 3.0, it is recommended that you specify the IDL class modifier **memory_management = corba** for the class. This will set **caller-owned** semantics for

Hello Application

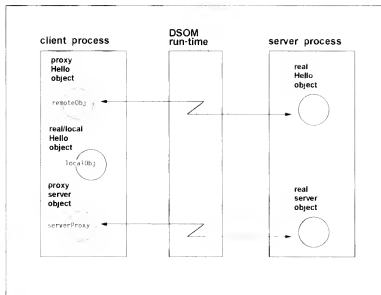


FIGURE 8.9 DSOM 2.x object/process layout with server object.

all parameters, and the only difference between DSOM 2.x and DSOM 3.0 will be **suppress_inout_free** for **inout** parameters. DSOM 2.x does not support reallocation of **inout** parameters. If a returned string is larger than the original input string, the result will be truncated to fit. In addition, DSOM 2.x does not support null pointer return values for **inout** parameters, which in DSOM 3.0 would cause deallocation of the original storage.

If you have defined **inout** parameters in DSOM 2.x, you should write your code to assume that the pointer value could change or be NULL on the return from the method call. Also ensure that the storage being passed is dynamically allocated, so that you don't run into deallocation problems with DSOM 3.0. The alternative is to specify **suppress_inout_free** for such parameters when you go to DSOM 3.0, but it would be fairly cumbersome and error prone to locate each such occurrence in your application, and DSOM will still allocate new storage on the return if the original storage is too small for a return string or if an object is being passed.

Parameter Passing

DSOM 2.x supports a limited set of parameter types for remote method calls. These are documented completely in the SOMObjects User's Guide in the section *Invoking Methods on Remote Objects*. The C++ types supported include the fundamental types, pointers, references, array, structures,

and classes. Note that pointers to pointers are not supported, except for the case of a `char**` or **SOMObject****, which map to a **string *** and a **SOM-Object *** in IDL respectively. Also, embedded pointers are not handled by DSOM 2.x, so you must handle the memory management for such parameters yourself.

Interlanguage Object Sharing

This chapter describes the DirectToSOM support available for two other languages, Smalltalk and OO-COBOL, and provides examples of using DirectToSOM C++ objects from these languages.

Introduction

In addition to the C++ language, DirectToSOM support is available for two other programming languages: OO COBOL and IBM Smalltalk. For DirectToSOM C++ and the C and C++ language bindings, the SOM compiler can generate the bindings directly from the IDL class description; however, the SOM compiler does not directly support these other languages. Instead, both the OO COBOL and Smalltalk products require that the IDL description for a class be registered in the interface repository. The OO COBOL compiler reads the interface repository directly as the program is compiled, while the Smalltalk product generates language bindings from the interface repository information, which can then be used by the programmer. Figure 9.1 shows the relationship between the various languages that are currently supported through SOM. The discussion and examples shown here are

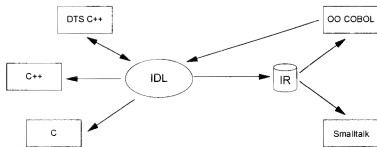


FIGURE 9.1 Language access to SOM class descriptions.

based on the IBM VisualAge for COBOL for OS/2 version 1.1 and the IBM VisualAge for Smalltalk for OS/2, version 2.0 products.

Smalltalk

The Smalltalk support for SOM, known as *SOMsupport*, is currently available as client-only starting with the IBM VisualAge for Smalltalk, version 2.0 product. The SOM compiler does not create language bindings for Smalltalk from IDL, rather *SOMsupport* uses the SOM interface repository to create native Smalltalk *wrapper classes* for specific SOM objects. Wrapper classes are generated explicitly through a framework of classes provided with the *SOMsupport* called the *SOM Smalltalk Constructor*. The wrapper classes can only be used to create and manipulate SOM objects as a client; there is no support for implementing SOM objects from within Smalltalk, either explicitly or implicitly by inheriting from one of the wrapper classes.

The generated wrapper classes have the same name as the SOM class, prefixed with the string *SOM*. In order to conform to Smalltalk naming conventions, underscores in class or method names are removed, and capitalization is changed (for example, the first letter of the class name or the letter after an underscore is capitalized). Once the wrapper classes are generated, they become part of the set of classes available to the application. These wrapper classes provide methods that can be invoked using standard Smalltalk syntax, but which map to calls to the *SOM API*. When an instance of a wrapper class is created, a corresponding *SOM class instance* is also created. Methods invoked on the wrapper instance result in the invocation of that method in the corresponding *SOM instance*.

Using the VisualAge SOMSupport

As an example of using a *SOM class* from within Smalltalk, the following shows a Smalltalk code sequence that creates a *SOM object* of *SOM class*

`Hello`, invokes the method `sayHello`, and deletes the object. Note that the class name is prepended with `SOM`. A `SOM` object is created in Smalltalk by sending the message `new` to the corresponding class object. The `SOM` model of class objects as run-time entities maps quite nicely to that of Smalltalk. Messages are sent to `SOM` objects using the standard Smalltalk syntax, as shown for `sayHello`. Because `SOM` objects exist outside the Smalltalk memory space, they are not garbage collected and must be explicitly freed through the `somFree` message.

```
| obj |
```

```
obj := SOMHello new.  
obj sayHello.  
obj somFree.
```

A major consideration in mapping from Smalltalk to `SOM` is memory management. When a Smalltalk object is no longer in use, it is automatically freed by the Smalltalk garbage collector. `SOM` objects exist outside the Smalltalk memory space, and thus must be destroyed explicitly through the **`SOMObject`** method `somFree`. Because the absolute memory address of a Smalltalk object can change as the memory manager reallocates storage, Smalltalk cannot pass the actual address of an object as a parameter to a `SOM` method. Instead, it copies the object to `SOM` memory and passes the address of the `SOM` memory copy. However, this copy exists only for the duration of the method call; hence, `SOM` implementation methods cannot reliably store this address and use it later. So if the object needs to be referenced in the future from within the implementation, a deep copy of it must be made by allocating storage for and copying nested substructures, rather than simply storing the address.

Setup

In order to enable Smalltalk `SOM` support you must first install `SOMObjects` for the VisualAge for Smalltalk product. From the System Transcript pull-down menu, select Smalltalk Tools, then Install. Once `SOMObjects` is installed for the VisualAge product, you can create wrapper classes for whatever `SOM` classes are available in the interface repository.

To simplify the process of creating wrapper classes, I created a class, `InitSom`, that provides a method, `generateWrapperForClass`, to generate wrappers for a given class. This class is shown next. In addition, I created a method, `unloadClass`, that unloads the DLL for a class from the Smalltalk program. This is required if the DLL needs to be re-created after Smalltalk has loaded it, otherwise the DLL will be locked by Smalltalk and cannot be updated. In order to use these methods, they must belong to a class that includes the **`SOMGlobals`** pool dictionary, as shown at line 4.

The `generateWrappersForClass` method accepts two parameters: the class name for which the wrapper should be generated and the target application for the generated wrappers. At line 13, the class definition is located in the interface repository. Line 15 instructs the SOM Smalltalk Constructor, **AbtSOMSmalltalkConstructor** to generate the class into the named application. Line 17 indicates that the class is not a DSOM class. Lines 18 and 19 create an ordered collection, `classDescriptions`, to which the wrapper class descriptions are added. Finally, lines 22 and 24 generate the wrapper classes into the target application. Note that it is possible to generate multiple classes at once—see the product documentation for details.

Smalltalk Setup Code (initsom.st):

```

1  Object subclass: #InitSom
2      instanceVariableNames: ''
3      classVariableNames: ''
4      poolDictionaries: 'SOMGlobals'
5
6  !InitSom class publicMethods !
7
8  generateWrapperForClass: className intoApplication:
9      intoApplicationName
10      "Generate wrapper for SOM class"
11      | interfaceDef classDescriptions |
12
13      interfaceDef := DefaultRepository
14          lookupId: className asRepositoryId.
15      AbtSOMSmalltalkConstructor
16          targetApplication: intoApplicationName.
17      AbtSOMSmalltalkConstructor remoteClass: false.
18      classDescriptions := OrderedCollection new.
19      interfaceDef abtSmalltalkConstructor
20          addClassDescriptionTo: classDescriptions.
21      interfaceDef somFree.
22      AbtSOMSmalltalkConstructor
23          generateNew: classDescriptions.
24      AbtSOMSmalltalkConstructor
25          generateTargetApplicationCode.
26
27  unloadClass: className
28      "Unload the given class file to free the dll"
29
30      SOMClassMgrInstance somUnloadClassFile: className.
31  !!
32
```

OO COBOL

OO COBOL extends COBOL-85 with support for object-oriented programming. There are currently no COBOL bindings for SOM, as for C++, such

that any OO COBOL compiler can access SOM. Instead, the IBM COBOL compilers provide DirectToSOM support by using SOM as the native object model for implementing the OO extensions. Native language syntax is used to define SOM objects, as with DirectToSOM C++, although the use of SOM is much more explicit with OO COBOL. Therefore, for the purposes of this discussion, language interoperability through SOM, the COBOL discussion is restricted to the support provided by IBM OO COBOL.

IBM OO COBOL supports a subset of the proposed ANSI OO COBOL definition [ANSI, 1994]. The spirit of the IBM OO COBOL definition is to provide essential OO building blocks through native support, and to take advantage of the existing support in the SOM API wherever possible. As such, the language description for these extensions is fairly brief. Native language support, based on the ANSI standard, is provided for describing classes and their methods, defining object variables, and method invocation. The object model itself, however, and most other object support is implemented using SOM. For example, objects are created and destroyed by invoking **SOMObject** and **SOMClass** class methods. The following is a brief overview of the native language support. Further details can be found in the product documentation.

Class Definition

New native language syntax is provided to define a class. All classes must directly or indirectly inherit from **SOMObject**, which implies that a class must always have at least one direct parent. Following the SOM model, a class is also an object with a metaclass of **SOMClass**; the class definition may designate a different class as the metaclass, but all metaclasses must derive from **SOMClass**. Multiple inheritance is supported, and the detailed semantics of inheritance are defined by SOM.

Instance data introduced in a class is accessible only by the methods introduced by that class, and is private otherwise. A class cannot access the instance data of a parent class or metaclass. There is no provision for defining class data that is shared across all classes through the class definition itself; this can be done through a metaclass, however.

IBM OO COBOL supports both the use and implementation of SOM objects. The compiler uses the SOM interface repository to extract information about referenced classes to perform compile-time static type checking, such as ensuring that a specified object type supports a given method and that the parameters and arguments are compatible. The compiler also supports generation of IDL definitions for SOM classes defined in the COBOL program. Thus a class implemented in C++, for example, can be used by an OO COBOL program, as can an OO COBOL class implementation be used from a Smalltalk program.

Methods

A class inherits all methods defined by its parent classes. There is no mechanism for hiding a method introduced by a parent class. A class may introduce new methods and override one introduced by a parent class to provide a different implementation. Name overloading by method signature is not supported; all methods introduced by a class must be uniquely named within that class, irrespective of case. When a parent method is overridden, the signatures of the overridden and overriding method must be compatible, which, depending upon the parameter type, means that they must be of the same class or one a parent class of the other. If the same method name is introduced by two different parent classes, the signatures for those methods must be compatible, and that of the leftmost class is used. This behavior can be modified by overriding the method in the new class and explicitly invoking the desired method by class name. Method binding is performed dynamically using name-lookup resolution.

There is no native support to automatically define CORBA attributes, but this can easily be achieved by introducing the appropriate `_get` and `_set` methods.

Initialization

No native support/syntax is provided for object creation and destruction. Instead, objects are created by invoking the **SOMClass** method `somNew` against a class object and destroyed through the **SOMObject** method `somFree`. Objects can be automatically initialized and deinitialized by overriding the **SOMObject** methods `somInit` and `somUninit`. (`somDefaultInit` and `somDestruct` are not used because OO COBOL does not provide support for the initialization and destruction control vectors.)

As required by the SOM model, a `<classname>NewClass` function is supplied by OO COBOL for all SOM classes defined within a COBOL program. As part of program initialization, the class initialization functions are implicitly called for any classes used by the COBOL program.

To illustrate the OO COBOL support, the following shows a simple example of a class definition, `Hello`, and a client written in OO COBOL. The two programs are compiled separately and then statically linked.

The class `Hello` is defined at line 2 in `hello.s.cb1` as inheriting from **SOMObject**. Every class that is used in the program must be declared in the **Repository** section, as shown for classes **SOMObject** and `Hello` at lines 6 and 7. This section may also be used to provide the names of the SOM class data structures, similar to the `DirectToSOM C++ SOMClassName` pragma. Methods are defined in the Identification Division of a class. The method `sayHello` is defined at lines 11 through 14. As with the C++ version, it simply displays a message and terminates.

The client program `hello.cbl` also contains a **Repository** section, but this time just for the class `Hello`, as that is the only class referenced in the program. At line 14, a new object of class `Hello` is created and assigned to the variable `obj`. The message `sayHello` is invoked against the object at line 16, and the object is deallocated at line 18. The compiler implicitly passes the target object as the first parameter to the method. The programs are compiled using the following command:

```
cob2 -qpgmname(longmixed) -qtypechk hello.cbl hellos.cbl
```

The **-qpgmname(longmixed)** option instructs the compiler to process program names as is, without truncation, folding, or translation. This is required for dealing with the longer, mixed-case, SOM names. The **-qtypechk** option instructs the compiler to perform static type checking of all method calls to ensure that the proper types are being passed.

Definition of COBOL Class Hello (hello/cbl/hellos.cbl):

```

1  Identification Division.
2  Class-id. Hello inherits SOMObject.
3  Environment Division.
4  Configuration section.
5  Repository.
6      Class SOMObject is "SOMObject"
7      Class Hello is "Hello".
8  Procedure Division.
9
10 Identification Division.
11 Method-id. sayHello.
12 Procedure Division.
13     Display "Hello from COBOL".
14 End method sayHello.
15
16 End class Hello.
```

Client of COBOL Class Hello (hello/cbl/hello.cbl):

```

1  Identification Division.
2  Program-id. "Client".
3  Environment Division.
4  Configuration section.
5  Repository.
6      Class Hello is "Hello".
7  Data Division.
8  Working-storage section.
9      01 obj usage object reference Hello.
10     01 env usage pointer.
11 Procedure Division.
12     Call "somGetGlobalEnvironment" returning env.
13     Display "Calling somNew".
```

Continued

```

14      Invoke Hello "somNew" returning obj
15      Display "Calling sayHello".
16      Invoke obj "sayHello".
17      Display "Calling somFree".
18      Invoke obj "somFree"
19      Goback.
20      End program "Client".

```

Style Guidelines for Defining DTS C++ Classes

This section provides some coding guidelines for designing DTS C++ classes and using them from other SOM-enabled languages, and serves partially to explain the coding practices used in the examples that follow. Much of the discussion is based on concepts covered in Chapter 6 and Chapter 7. Refer to these chapters for background information.

Class and Member Names

One of the major considerations with interlanguage sharing of DTS C++ classes is name mangling. As discussed in Chapter 7, because IDL is case-insensitive and does not support name overloading within a class, C++ member and class names are mangled to provide unique IDL names. In order to accommodate this, C++ names are mangled using a SOM mangling scheme that loosely follows the mangling scheme of most C++ compilers. In addition, uppercase characters are converted to lowercase by prepending them with a lowercase *z*; *z_* is used to mean a real lowercase *z*. So, *somSayHelloZz* becomes *somzSayzhellozzz_*.

When working only within DTS C++, this does not pose a problem; however, the mangled names tend to be fairly long and unreadable, making them unsuitable for use from other languages. There are several pragmas that can be used to affect DTS C++ name mangling for SOM: **SOMNoMangling**, **SOMMethodName**, **SOMDataName**, and **SOMClassName**.

SOMNoMangling prevents the name mangling of class member names and can be turned on for a specific class or for all classes in the compilation unit. If the class has overloaded member functions, this causes collisions in the generated IDL, in which case the **SOMMethodName** pragma is also required to give specific names to overloaded members. **SOMDataName** can be used to give specific names to class data members. **SOMNoMangling** does not affect class names, which are at the very least mangled to be case-insensitive by translating uppercase letters to their lowercase equivalents preceded by lowercase *z*. Template class names are mangled further to incorporate type information. Therefore, if the class name contains uppercase letters or is a template class, the **SOMClassName** pragma should also be used to ensure that the class name is not mangled.

Note that IDL matches overloaded methods by name only, so if a name is not mangled and is subsequently mangled, this will break binary compatibility because it is equivalent to changing the method name, or removing the method with the old name and adding a method with the new name. Methods cannot be deleted from a SOM class without potentially breaking binary compatibility. If interlanguage sharing will likely be required, the recommended approach is to always use **SOMNoMangling** and **SOMClassName**, and to use **SOMMethodName** when necessary to handle overloaded member names or special names such as operators.

Irrespective of mangling, names should contain only alphabetic characters or digits, and begin with an alphabetic character. Underscores, while valid for IDL names, may cause problems in some languages, such as Smalltalk, that remove them from names.

Data Members

Most other languages do not have the capability to directly access public or protected data members or static data members. The simplest way to allow other languages to access public data members is by making them into CORBA attributes. This can be done using the **SOMAttribute** pragma. The DTS C++ compiler implicitly generates **_get/_set** methods that return and set the member value respectively. By default, the backing data becomes private and all C++ access to the data members outside the class is through the attribute functions; so for performance reasons, the backing data could be made public for direct access from DTS C++.

For public static data members, which are shared across all class instances, the recommended approach is to define a metaclass for the given class and make the static data member an attribute of the metaclass.

Constructors

SOM requires that a class define a default constructor with no arguments, which is mapped to an override of the **SOMObject** method **somDefaultInit**. As discussed in Chapter 6, *Inside DirectToSOM C++*, C++ copy constructors override one of the **SOMObject** methods **somDefaultCopyInit**, **somDefaultConstCopyInit**, **somDefaultVCopyInit**, or **somDefaultConstVCopyInit**. Any other C++ constructors have mangled names and should be given SOM names explicitly via the **SOMMethodName** pragma.

Through both OO COBOL and Smalltalk, **somNew** can be invoked to create objects. **somNew** implicitly calls **somDefaultInit** as part of object creation, which in turn calls the C++ no-argument default constructor. This can also be performed in two steps: first by creating an uninitialized object through invoking the **SOMClass** methods **somNewNoInit** or **somRenewNoInit** against a given class object, and then by invoking

somDefaultInit against the object. This two-step mechanism can be used to call copy constructors, such as **somDefaultCopyInit** or other constructors, from OO COBOL or Smalltalk, which don't have language mechanisms to implicitly call these methods. All **SOMObject** constructor methods accept two parameters: the target object and an initialization control vector. (Recall from Chapter 6 that the **SOMObject** methods are **oidl**, and use the global **Environment** rather than being passed an **Environment** parameter.) The initialization control vector is used to prevent a class constructor from being called more than once when a class appears multiple times in the inheritance tree, and should always be supplied as a null pointer. If a constructor is called directly, a null pointer must be supplied for the initialization control vector parameter.

Assignment

If an **operator=** method is not defined for the class, the compiler supplies four SOM assignment methods, one of which is called when an assignment operator is encountered. These methods are **somDefaultAssign**, **somDefaultConstAssign**, **somDefaultVAssign**, and **somDefaultConstVAssign**. If an **operator=** method is supplied, then it is called when one of the SOM assignment methods is invoked. The SOM assignment methods accept a first parameter that is an assignment control vector, followed by the source object. As with the initialization control vector, the assignment control parameter is always passed as a null pointer and is used to prevent a base from being assigned more than once when it appears multiple times in the hierarchy. This support is not available for **operator=**. For this reason, the SOM assignment methods should be defined rather than an **operator=** method.

Examples

This section provides examples of sharing code with DTS C++, OO COBOL, and Smalltalk under OS/2 Warp with SOMObjects Release 2.1. (Note: These programs were tested using IBM VisualAge C++ version 3.0, IBM VisualAge for COBOL for OS/2 version 1.1, and IBM VisualAge for Smalltalk for OS/2 version 2.0 at CSD level 1. CSD 1, or version 3.0 of the Smalltalk product is necessary in order to run the second and third examples.) There are three sets of examples. The first is a very simple class to test invoking one method; the second illustrates parameter passing with methods; and the third uses more complicated C++ constructs.

A Simple Example

The first example is a simple DTS C++ class **Hello**. Note the use of the **SOMNoMangling** and **SOMClassName** pragmas in the class definition file

at lines 3 and 6 to control IDL name generation. The **SOMIDLPass** pragma is used at line 7 to indicate the name of the DLL that contains the current class. The DLL name is used by the SOM API when a class is dynamically loaded at run time rather than statically bound to the application. While SOM classes should be designed for both situations, it is particularly important for this application, as Smalltalk will dynamically load any required DLLs. The file `tsHello.cpp` shows how this class is used in C++ using standard C++ syntax. This is fairly standard in the examples throughout the book so far.

The OO COBOL client program is very similar to the one shown earlier, which used a COBOL class implementation. First, the **somNew** method is invoked against the class object `Hello`, which returns an instance of the class `Hello`. Then the `sayHello` message is sent to the object, and finally the object is deleted via **somFree**. The same approach is followed for Smalltalk. For the Smalltalk client, the SOM object must also be explicitly freed as in the OO COBOL program; the Smalltalk garbage collector does not handle SOM objects. The only difference between this and the earlier COBOL client is the explicit passing of an **Environment** parameter to the method `sayHello`. OO COBOL requires that the CORBA **Environment** parameter be passed explicitly to the target method. This parameter is passed implicitly by the compiler for both DTS C++ and Smalltalk.

Definition of DirectToSOM C++ Class Hello (hello/cpp/hello.hh):

```

1  #include <som.hh>
2
3  #pragma SOMNOMangling(on)
4
5  class Hello : public SOMObject {
6      #pragma SOMClassName(*, "Hello")
7      #pragma SOMIDLPass(*, "Implementation-End", \
8          *dlname = \"hello.dll\";*)
9  public:
10     void sayHello();
11 };

```

Implementation of DirectToSOM C++ Class Hello (hello/cpp/hello.cpp):

```

1  #include <fstream.h>
2  #include "hello.hh"
3
4  void Hello::sayHello()
5  {
6      cout << "Hello from C++" << endl;
7  }
8
9  SOMEXTERN void SOMLINK EXPORT SOMInitModule(
10     long majorVersion, long minorVersion, string className)
11 {

```

Continued

```

12     HelloNewClass(majorVersion, minorVersion);
13 }

```

C++ Client of DirectToSOM C++ Class Hello (hello/cpp/tsthello.cpp):

```

1  #include <iostream.h>
2  #include "hello.hh"
3
4  int main(void)
5  {
6      Hello obj;
7
8      obj.sayHello();
9  }

```

COBOL Client of DirectToSOM C++ Class Hello (hello/cbl/hello.cbl):

```

1      Identification Division.
2      Program-id. "Client".
3      Environment Division.
4      Configuration section.
5      Repository.
6          Class Hello is "Hello".
7      Data Division.
8      Working-storage section.
9          01 obj usage object reference Hello.
10         01 env usage pointer.
11      Procedure Division.
12          Call "somGetGlobalEnvironment" returning env.
13          Display "Calling somNew".
14          Invoke Hello "somNew" returning obj
15          Display "Calling sayHello".
16          Invoke obj "sayHello" using by value env.
17          Display "Calling somFree".
18          Invoke obj "somFree"
19          Goback.
20      End program "Client".

```

Smalltalk Client of DirectToSOM C++ Class Hello (hello/st/hello.st):

```

1      | obj |
2
3      obj := SOMHello new.
4      obj sayHello.
5      obj somFree.

```

Makefile for C++ (hello/cpp/makefile):

```

1  all: hello.idl hello.dll tsthello.exe
2
3  ICCOPTS = -DEXPORT=__Export
4
5  hello.dll: hello.hh hello.cpp hello.def
6      icc $(ICCOPTS) /Ti+ /Ge- hello.cpp hello.def

```



```

7      implib hello.lib hello.dll
8
9  tsthello.exe: hello.hh tsthello.cpp
10     gcc $(ICCOPTS) /Ti+ tsthello.cpp hello.lib
11
12 hello.idl: hello.hh
13     gcc hello.hh
14     sc -sir -u hello.idl
15     sc -sdef hello.idl

```

Generated IDL for DirectToSOM C++ Class Hello:

```

1  #ifndef __hello_idl
2  #define __hello_idl
3  /*
4  *
5  * Generated on Mon Apr 22 11:59:51 1996
6  * Generated from hello.hh
7  * Using IBM VisualAge C++ for OS/2, Version 3
8  */
9  #include <somobj.idl>
10 interface Hello;
11 interface Hello : SOMObject {
12     void sayHello ();
13 #ifdef __SOMIDL__
14     implementation {
15         align=0;
16         sayHello: public,nonstatic,cxxmap="sayHello()",
17             cxxdecl="void sayHello();";
18         somDefaultConstVAssign: public,override;
19         somDefaultConstAssign: public,override;
20         somDefaultConstVCopyInit: public,override,init;
21         somDefaultInit: public,override,init;
22         somDestruct: public,override;
23         somDefaultCopyInit: public,override;
24         somDefaultConstCopyInit: public,override;
25         somDefaultVCopyInit: public,override;
26         somDefaultAssign: public,override;
27         somDefaultVAssign: public,override;
28         declarationorder = "sayHello, somDefaultConstVAssign,
29             somDefaultConstAssign, somDefaultConstVCopyInit,
30             somDefaultInit, somDestruct";
31         releaseorder:
32             sayHello;
33         callstyle = idl;
34         dtsclass;
35         directinitclasses = "SOMObject";
36         cxxmap = "Hello";
37         cxxdecl = "class Hello : public virtual SOMObject";
38         dllname = hello;
39     };
40 #endif
41 };
42 #endif /* __hello_idl */

```

Passing Parameters

The next example illustrates parameter passing and the use of CORBA attributes. The class `Parms` has three data members: an integer `i`, a floating point number `f`, and a string `s`. The **SOMAttribute** pragma is applied to the member `i` at line 8 and the member `f` at line 10. This will cause the compiler to implicitly define and generate the methods `_get` and `_set` that retrieve and update the values of the member respectively. This allows C++ data members to be accessed from other languages, as shown in the Smalltalk and OO COBOL client programs. However, these data members can still be accessed using data member syntax in C++, as shown in the C++ client program.

Rather than making the string member an attribute, which would certainly work, I instead chose to provide separate methods to set and get the string to illustrate the use of default and **inout** parameters, and to provide more language-neutral forms of string handling. The `setString` method accepts a required string parameter and an optional length parameter. If the length parameter is not supplied, or is supplied with `-1`, the string is assumed to be a null-terminated string. Otherwise, the length parameter supplies the length of the string to copy. The `getString` method is used to retrieve a null-terminated string, whereas the `getString2` method will return a string and the length of that string. The `long*` parameter will be generated as **inout**, as shown at line 16 in the generated IDL file.

OO COBOL has no direct support for attributes, but these methods can simply be called directly using their method names, as shown in the OO COBOL client at lines 29 and 30. IDL string types should be passed in OO COBOL as a **by value** pointer, rather than a **by reference** character, as shown at line 42 for `msgp`. Otherwise, the **typechk** option will flag an error. For the **inout** length parameter to `getString2`, I passed `i` by reference at line 44. This will pass the address of `i` to the method. This program also illustrates error handling with the **Environment** parameter. If the call to **somNew** at line 23 is not successful, the error-handling code at lines 51 through 58 will be invoked. Because the **SOMObject** methods are **oidl**, they do not use a passed **Environment** parameter for supplying error information. Instead, they use the global **Environment** structure.

Smalltalk generates the `get` and `set` methods for attributes in the Smalltalk style of `member` and `member:`, which is illustrated in the Smalltalk client at lines 6 and 8 for attribute `i`. In Smalltalk, **in** parameters are just passed directly, but **out** and **inout** parameters are passed as arrays where the first element in the array contains the resulting value. The methods **asOUTParameter** and **asINOUTParameter** can be used to create an array containing one object; **asINOUTParameter** is used at line 18 in the Smalltalk client to retrieve the length result when calling `getString2`.

Definition of DirectToSOM C++ Class *parms* (*parms/cpp/parms.hh*):

```

1  #include <som.hh>
2
3  #pragma SOMNoMangling(on)
4  class Params : public SOMObject {
5      char s[256];
6      public:
7          long i;
8          #pragma SOMAttribute(i)
9          double f;
10         #pragma SOMAttribute(f)
11         void setString(char *, long = -1);
12         char *getString();
13         char *getString2(long*);
14         #pragma SOMReleaseOrder(i, f, \
15             setString, getString, getString2)
16         #pragma SOMClassName(*, "Params")
17         #pragma SOMIDLPass(*, "Implementation-End", \
18             "dllname = \"parms.dll\";")
19     };

```

Implementation of DirectToSOM C++ Class *Params* (*parms/cpp/parms.cpp*):

```

1  #include <fstream.h>
2  #include "parms.hh"
3
4  void Params::setString(char *in, long length)
5  {
6      if (length == -1)
7          strcpy(s, in);
8      else {
9          strncpy(s, in, length);
10         s[length] = '\0';
11     }
12     cout << "storeString(string): s is " << s <<
13         ", length is " << length << endl;
14 }
15
16 char *Params::getString()
17 {
18     cout << "getString(string)" << endl;
19     return s;
20 }
21
22 char *Params::getString2(long *length)
23 {
24     cout << "getString(string)*" << endl;
25     *length = strlen(s);
26     return s;
27 }

```

Continued

```

28
29 SOMEXTERN void EXPORT SOMLINK SOMinitModule(
30     long majorVersion, long minorVersion, string className)
31 {
32     ParamsNewClass(0, 0);
33 }

```

C++ Client of DirectToSOM C++ Class Params (params/cpp/tstparams.cpp):

```

1  #include <iostream.h>
2  #include "params.hh"
3
4  int main(void)
5  {
6      Params obj;
7
8      cout << "setting i to 6" << endl;
9      obj.i = 6;
10     cout << "i is " << obj.i << endl;
11
12     cout << "setting f to 99.78" << endl;
13     obj.f = 99.78;
14     cout << "f is " << obj.f << endl;
15
16     obj.setString("String from C++");
17     cout << "string is " << obj.getString() << endl;
18 }

```

OO COBOL Client of DirectToSOM C++ Class Params (params/cbl/params.cbl):

```

1      Identification Division.
2      Program-id. "Client".
3      Environment Division.
4      Configuration section.
5      Repository.
6          Class Params is "Params".
7      Data Division.
8      Working-storage section.
9          01 obj usage object reference Params.
10         01 env usage pointer.
11         01 i pic s9(9) usage binary.
12         01 f usage comp-2.
13         01 msg pic X(17) value "String from COBOL".
14         01 msgp usage pointer.
15         01 txtp usage pointer.
16      Linkage section.
17          77 msg2 pic X(20).
18         01 txt2 pic X(25).
19         01 envMaj pic s9(9) usage binary.
20      Procedure Division.
21          Call "somGetGlobalEnvironment" returning env.
22          Set Address of envMaj to env.

```

```

23      Invoke Params 'somNew' returning obj.
24      If envMaj not = 0
25          Perform error-handler.
26
27      Display "Calling set_i with 5".
28      move 5 to i.
29      Invoke obj '_set_i' using by value env i.
30      Invoke obj '_get_i' using by value env
31          returning i.
32      Display "_get_i returns: " i.
33      move 27.5 to f.
34      Display "Calling set_f with " f.
35      Invoke obj '_set_f' using by value env f.
36      Invoke obj '_get_f' using by value env
37          returning f.
38      Display "_get_f returns: " f.
39      move length msg to i.
40      set msgp to address of msg.
41      Invoke obj 'setString' using
42          by value env by value msgp by value i.
43      Invoke obj 'getString2' using
44          by value env by reference i returning msgp.
45      Set Address of msg2 to msgp.
46      Display "getString2 returns: "
47          msg2(1:i) ", length: " i.
48      Invoke obj 'somFree'.
49      Goback.
50
51  error-handler.
52      call 'somExceptionId' using by value env
53          returning txtp.
54      Set Address of txt2 to txtp.
55      Display "major: " envMaj.
56      Display "error: " txt2.
57      call "somExceptionFree" using by value env.
58      Goback.
59
60  End program "Client".

```

Smalltalk Client of DirectToSOM C++ Class Params (parms/st/parms.st):

```

1      | obj msg msg2 length |
2
3      obj := SOMParams new.
4      Transcript cr; show: 'Setting i to 5'.
5
6      obj i: 5.
7      Transcript cr; show: 'i is: '.
8      obj i printOn: Transcript.
9
10     obj f: 66.7.
11     Transcript cr; show: 'Setting f to 66.7'.
12     Transcript cr; show: 'f is: '.
13     obj f printOn: Transcript.
14

```

Continued

```

15     "use length-dependent strings"
16     msg := 'Hello from Smalltalk'.
17     obj setString: msg pArg2: msg size.
18     msg2 := obj getString2: (length := 0 asINOUTParameter).
19     Transcript cr; show: 'string is: ', msg2, ', length: '.
20     (length at: 1) printOn: Transcript.
21
22     "use null-terminated strings"
23     msg := 'Hello from Smalltalk'.
24     obj setString: msg pArg2: -1.
25     msg2 := obj getString.
26     Transcript cr; show: 'string is: ', msg2, ', length: '.
27     msg2 size printOn: Transcript.
28
29     obj somFree.

```

Makefile for C++ (parms/cpp/makefile):

```

1  all: parms.idl parms.dll tstparms.exe
2
3  ICCOPTS = -DEXPORT=_Export
4
5  parms.dll: parms.hh parms.cpp parms.def
6      icc $(ICCOPTS) /Ti+ /Ge- parms.cpp parms.def
7      implib parms.lib parms.dll
8
9  tstparms.exe: parms.hh tstparms.cpp
10      icc $(ICCOPTS) /Ti+ tstparms.cpp parms.lib
11
12  parms.idl: parms.hh
13      icc parms.hh
14      sc -sir -u parms.idl
15      sc -sdef parms.idl

```

Generated IDL for DirectToSOM C++ Class Params:

```

1  #ifndef __parms_idl
2  #define __parms_idl
3  /*
4   *
5   * Generated on Mon Apr 22 13:56:38 1996
6   * Generated from parms.hh
7   * Using IBM VisualAge C++ for OS/2, Version 3
8   */
9  #include <somobj.idl>
10 interface Params;
11 interface Params : SOMObject {
12     attribute long i;
13     attribute double f;
14     void setString (in string p_arg1, in long p_arg2);
15     string getString ();
16     string getString2 (inout long p_arg1);
17 #ifdef __SOMIDL__

```

```

18     implementation {
19         align=4;
20         char s[256];
21         s: cxxmap="s",offset=0,align=1,size=256,
22             nonstaticaccessors,private,
23             cxxdecl="char s[256];";
24         i: cxxmap="i",offset=256,align=4,size=4,
25             nonstaticaccessors,private,publicaccessors,
26             cxxdecl="long i;";
27         f: cxxmap="f",offset=260,align=8,size=8,
28             nonstaticaccessors,private,publicaccessors,
29             cxxdecl="double f;";
30         setString: public,nonstatic,
31             cxxmap="setString(char*,long)",
32             cxxdecl="void setString(char*,long = -1);";
33         getString: public,nonstatic, cxxmap="getString()",
34             cxxdecl="char* getString();";
35         getString2: public,nonstatic,
36             cxxmap="getString2(long*)",
37             cxxdecl="char* getString2(long*);";
38         somDefaultConstVAssign: public,override;
39         somDefaultConstAssign: public,override;
40         somDefaultConstVCopyInit: public,override,init;
41         somDefaultInit: public,override,init;
42         somDestruct: public,override;
43         somDefaultCopyInit: public,override;
44         somDefaultConstCopyInit: public,override;
45         somDefaultVCopyInit: public,override;
46         somDefaultAssign: public,override;
47         somDefaultVAssign: public,override;
48         declarationorder = "s, i, f, setString, getString,
49             getString2, somDefaultConstVAssign,
50             somDefaultConstAssign, somDefaultConstVCopyInit,
51             somDefaultInit, somDestruct";
52         releaseorder:
53             _get_i,
54             _set_i,
55             _get_f,
56             _set_f,
57             setString,
58             getString,
59             getString2,
60     #ifdef __PRIVATE__
61         s,
62     #else
63         s__P0,
64     #endif
65         i,
66         f;
67         callstyle = idl;
68         dtsclass;
69         directinitclasses = "SOMObject";
70         cxxmap = "Parms";
71         cxxdecl = "class Parms : public virtual SOMObject";

```

Continued

```

72         dllname = parms;
73     };
74 #endif
75 };
76 #endif /* __parms_idl */

```

Invoking Constructor and Assignment Methods

The final example illustrates the use of a range of C++ class methods, such as constructors and assignment operators. The class `GenericString` contains a default constructor at line 11 (maps to **somDefaultInit**), a copy constructor at line 12 (**somDefaultConstCopyInit**) and a third non-SOM constructor at line 13 that is given a SOM name of `setInit` at line 14. Also defined for the class `GenericString` is **somDefaultConstAssign**, which is called for **operator=** in the C++ client.

The client programs show how these methods are used in C++, OO COBOL, and Smalltalk. In addition to using the methods statically, the C++ client also invokes the methods dynamically, by resolving to the methods using the **somResolveByName** method, at lines 38 and 44.

Recall from Chapter 6 that each of the 10 **SOMObject** methods, plus any additional initializer methods, are implicitly passed a control structure, which is passed as null when the initializer is called from the client program. The DirectToSOM C++ compiler implicitly passes null when calling these routines from client code, but other languages must explicitly pass the null control vector value. For example, in the OO COBOL client program at line 47, **somDefaultConstCopyInit** is passed a first parameter of `nullp`. This is a pointer with a value of null, as declared at line 17.

In the Smalltalk example, it was necessary to retrieve the initialization and assignment control vectors and pass them to the methods at lines 18 and 25, rather than just pass a null pointer. This is because, as discussed in Chapter 7, the IDL for the initialization and assignment methods originally defined the control vector parameters as **inout** rather than **in**. The SOM-supplied IDL files have since been changed in a CSD for SOMObjects 2.1, but this will not be fixed until the next release of the VisualAge C++ product. The IDL generated by the DirectToSOM C++ compiler I used (version 3.0) generated **inout** rather than **in**, as shown at line 14 in the generated IDL file. Smalltalk enforces the CORBA rule in not allowing a null pointer to be passed for an **inout** parameter. This is because the target method may depend upon the **inout** parameter containing valid address values. It is therefore important to always provide a valid address for an **inout** parameter, particularly if you don't have access to the source code. If the version of the compiler that you are using generates **in** rather than **inout**, you do not need to perform the extra step of retrieving the control vector, and can simply pass null.

Definition of DirectToSOM C++ Class GenericString (string/cpp/genstr.hh):

```

1  #include <som.hh>
2
3  #pragma SOMNoMangling(on)
4
5  class GenericString : public SOMObject {
6  public:
7      long length;
8      #pragma SOMAttribute(length, readonly)
9      char *data;
10     #pragma SOMAttribute(data, readonly)
11     GenericString();
12     GenericString(const GenericString&);
13     GenericString(char *, long = -1);
14     #pragma SOMMethodName(\
15         GenericString(char *, long), "setInit")
16     GenericString& set(char *, long = -1);
17     SOMObject *somDefaultConstAssign(
18         somAssignCtrl *, const SOMObject*);
19     void clear();
20     void display();
21     ~GenericString();
22     #pragma SOMClassName(*, "GenericString")
23     #pragma SOMIDLPass(*, "Implementation-End", \
24         "dllname = \"genstr.dll\";")
25     #pragma SOMReleaseOrder( \
26         length, data, \
27         GenericString(char *, long), \
28         set(char *, long), \
29         clear(), display())
30 };
31

```

Implementation of DirectToSOM C++ Class GenericString (string/cpp/genstr.cpp):

```

1  #include <stdlib.h>
2  #include <iostream.h>
3  #include "genstr.hh"
4
5  GenericString::GenericString()
6  {
7      length = 0;
8      data = NULL;
9  }
10
11 GenericString::GenericString(const GenericString &from)
12 {
13     length = 0;
14     data = NULL;

```

Continued

```

15     *this = from;
16     return;
17     length = from.length;
18     data = new char[length + 1];
19     strncpy(data, from.data, length);
20     data[length] = '\0';
21 }
22
23 SOMObject *GenericString::somDefaultConstAssign(
24     somAssignCtrl *ctrl, const SOMObject* fromObject)
25 {
26     const GenericString *fromString =
27         (GenericString *)fromObject;
28     length = fromString->length;
29     if (data)
30         delete data;
31     data = new char[length + 1];
32     strncpy(data, fromString->data, length);
33     data[length] = '\0';
34     return this;
35 }
36
37 GenericString::GenericString(
38     char *initialData, long initialLength)
39 {
40     length = 0;
41     data = NULL;
42     set(initialData, initialLength);
43 }
44
45 GenericString &GenericString::set(
46     char *initialData, long initialLength)
47 {
48     if (initialLength < -1 || initialLength > 50)
49         length = 5;
50     else if (initialLength == -1)
51         length = strlen(initialData);
52     else
53         length = initialLength;
54     if (data)
55         delete data;
56     data = new char[length + 1];
57     strncpy(data, initialData, length);
58     data[length] = '\0';
59     return *this;
60 }
61
62 void GenericString::clear()
63 {
64     length = 0;
65     if (data)
66         delete data;
67     data = NULL;
68 }
69

```

```

70 void GenericString::display()
71 {
72     cout << "    data: " << data
73         << ", length: " << length << endl;
74 }
75
76 GenericString::~GenericString()
77 {
78     if (data)
79         delete data;
80 }
81
82
83 SOMEXTERN void SOMLINK EXPORT SOMInitModule(
84     long major, long minor, char *name)
85 {
86     GenericStringNewClass(major, minor);
87 }

```

C++ Client of DirectToSOM C++ Class GenericString (string/cpp/tststr.cpp):

```

1  #include "genstr.hh"
2  #include <iostream.h>
3
4  int main(void)
5  {
6      cout << "create s1 statically with "
7          << "default constructor" << endl;
8      GenericString s1;
9      s1.display();
10
11     cout << "set s1 string value" << endl;
12     s1.set("string1");
13     s1.display();
14
15     cout << "create s2 statically with "
16         << "string constructor" << endl;
17     GenericString s2("string2");
18     s2.display();
19
20     cout << "create s3 statically with "
21         << "copy constructor from s2" << endl;
22     GenericString s3(s2);
23     s3.display();
24
25     cout << "clear s3" << endl;
26     s3.clear();
27     s3.display();
28
29     cout << "assign s1 to s3" << endl;
30     s3 = s1;
31     s3.display();
32

```

Continued

```

33     cout << "create obj dynamically with "
34           "string constructor" << endl;
35     GenericString *obj = (GenericString *)
36         GenericString::_ClassObject->somNewNoInit();
37     typedef void (*mpt) (...);
38     mpt mp = (mpt)somResolveByName(obj, "setInit");
39     mp(obj, somGetGlobalEnvironment(), 0, "obj", 7);
40     obj->display();
41
42     cout << "create obj2 dynamically with "
43           "copy constructor from s2" << endl;
44     mp = (mpt)somResolveByName(obj, "somDefaultConstCopyInit");
45     GenericString *obj2 = (GenericString *)
46         GenericString::_ClassObject->somNewNoInit();
47     mp(obj2, 0, s2);
48     obj2->display();
49
50     delete obj;
51     delete obj2;
52 }

```

***OO COBOL Client of DirectToSOM C++ Class GenericString
(string/cbl/genstr.cbl):***

```

1      Identification Division.
2      Program-id. "Client".
3      Environment Division.
4      Configuration section.
5      Repository.
6          Class GenericString is "GenericString".
7      Data Division.
8      Working-storage section.
9          01 env usage pointer.
10         01 str1 usage object reference GenericString.
11         01 str2 usage object reference GenericString.
12         01 str3 usage object reference GenericString.
13         01 objp usage object reference GenericString.
14         01 txt pic X(25) value "string1 from COBOL".
15         01 len pic s9(9) comp.
16         01 p usage pointer.
17         01 nullp usage pointer value null.
18         01 somNewNoInit pic x(12) value "somNewNoInit".
19         01 mp usage procedure-pointer.
20         01 b pic s9(9) binary.
21      Linkage section.
22         01 txt2 pic X(25).
23         01 envMaj pic s9(9) usage binary.
24      Procedure Division.
25         Call "somGetGlobalEnvironment" returning env.
26         Set Address of envMaj to env.
27         Display "Calling somNew".
28         Invoke GenericString "somNew" returning str1
29         If envMaj not = 0
30             Perform error-handler.

```

```

31
32     move length of txt to len.
33     set p to address of txt.
34     Display "Calling Set".
35     Invoke str1 "set" using by value env
36         by value p by value len returning objp.
37     Display "Calling _get_data".
38     Invoke str1 "_get_data" using
39         by value env returning p.
40     Set Address of txt2 to p.
41     Display txt2.
42
43     Display "Calling somNewNoInit".
44     Invoke GenericString "somNewNoInit"
45         returning str2.
46     Display "Calling somDefaultConstCopyInit".
47     Invoke str2 "somDefaultConstCopyInit" using
48         by value nullp str1.
49     Invoke str2 "display" using by value env.
50
51     Move "assign string" to txt.
52     set p to address of txt.
53     Invoke str1 "Set" using by value env
54         by value p by value len returning objp.
55     Display "Calling somDefaultAssign".
56     Invoke str2 "somDefaultAssign" using
57         by value nullp str1 returning objp.
58     Invoke str1 "display" using by value env.
59
60     Display "Calling somNewNoInit".
61     Invoke GenericString "somNewNoInit"
62         returning str3.
63
64     Move "setInit string" to txt.
65     set p to address of txt.
66     move 14 to b.
67     Display "Calling setInit".
68     Invoke str3 "setInit" using
69         by value env nullp p b.
70     Display "Calling display".
71     Invoke str3 "display" using by value env.
72
73     Display "Calling somFree".
74     Invoke str1 "somFree".
75     Invoke str2 "somFree".
76     Invoke str3 "somFree".
77     Goback.
78
79     error-handler.
80     call "somExceptionId" using by value env
81         returning p.
82     Set Address of txt2 to p.
83     Display "major: " envMaj.
84     Display "error: " txt2.

```

Continued

```

85         call "somExceptionFree" using by value env.
86         Goback.
87     End program "Client".

```

Smalltalk Client of C++ DirectToSOM Class GenericString (string/st/string.st):

```

1      | obj1 obj2 obj3 ctrl |
2
3      self halt.
4      obj1 := SOMGenericString new.
5      obj1 set: 'Smalltalk object 1' pArg2: -1.
6      Transcript cr; show:
7          'After init set: obj1 data is ''',
8          obj1 data, ''.
9
10     ctrl := Array new: 1.
11     SOMGenericString somGetInstanceInitMask: ctrl.
12     obj2 := SOMGenericString somNewNoInit.
13     obj2 somDefaultCopyInit: ctrl fromObj: obj1.
14     Transcript cr; show:
15         'After somDefaultCopyInit obj2 data is ''',
16         obj2 data, ''.
17
18     SOMGenericString somGetInstanceAssignmentMask: ctrl.
19     obj2 set: 'Assigned from Smalltalk object 2' pArg2: -1.
20     obj1 somDefaultConstAssign: ctrl fromObj: obj2.
21     Transcript cr; show:
22         'After somDefaultConstAssign obj1 data is ''',
23         obj1 data, ''.
24
25     SOMGenericString somGetInstanceInitMask: ctrl.
26     obj3 := SOMGenericString somNewNoInit.
27     self halt.
28     obj3 parms: 1 pArg2: 2 pArg3: 3 pArg4: 4 pArg5: 5 pArg6: 6.
29
30     obj3 setInit: ctrl pArg1: 'Smalltalk object 3' pArg2: -1.
31     Transcript cr; show:
32         'After setInit obj3 data is ''', obj3 data, ''.

```

C++ Makefile (string/cpp/makefile):

```

1  all: genstr.idl genstr.dll tststr.exe
2
3  ICCOPTS = -DEXPORT=_Export
4
5  genstr.dll: genstr.hh genstr.cpp genstr.def
6      gcc $(ICCOPTS) /Ti+ /Ge- genstr.cpp genstr.def
7      implib genstr.lib genstr.dll
8
9  tststr.exe: genstr.hh tststr.cpp
10      gcc $(ICCOPTS) /Ti+ tststr.cpp genstr.lib
11

```

```

12 genstr.idl: genstr.hh
13         icc genstr.hh
14         sc -sir -u genstr.idl
15         sc -sdef genstr.idl

```

Generated IDL for Class GenericString:

```

1  #ifndef __genstr_idl
2  #define __genstr_idl
3  /*
4   *
5   * Generated on Mon Apr 22 15:17:01 1996
6   * Generated from genstr.hh
7   * Using IBM VisualAge C++ for OS/2, Version 3
8   */
9  #include <somobj.idl>
10 interface GenericString;
11 interface GenericString : SOMObject {
12     readonly attribute long length;
13     readonly attribute string data;
14     void setInit (inout somInitCtrl ctrl,
15                 in string p_arg1, in long p_arg2);
16     GenericString set (in string p_arg1, in long p_arg2);
17     void clear ();
18     void display ();
19 #ifdef __SOMIDL__
20     implementation {
21         align=4;
22         length: cxxmap="length",offset=0,align=4,size=4,
23                 nonstaticaccessors,private,publicaccessors,
24                 cxxdecl="long length;";
25         data: cxxmap="data",offset=4,align=4,size=4,
26               nonstaticaccessors,private,publicaccessors,
27               cxxdecl="char* data;";
28         somDefaultInit: public,override,init,
29                 cxxdecl="GenericString();";
30         somDefaultConstCopyInit: public,override,init,
31                 cxxdecl="GenericString(const GenericString&);";
32         setInit: public,nonstatic,init,
33                 cxxmap="GenericString(char*,long)",
34                 cxxdecl="GenericString(char*,long = -1);";
35         set: public,nonstatic,cxxmap="set(char*,long)",
36               cxxdecl="GenericString& set(char*,long = -1);";
37         somDefaultConstAssign: public,override, cxxdecl="\
38             \"virtual SOMObject* somDefaultConstAssign\"
39             \"(somAssignCtrl*,const SOMObject*);\";
40         clear: public,nonstatic,cxxmap="clear()",
41               cxxdecl="void clear();";
42         display: public,nonstatic,cxxmap="display()",
43               cxxdecl="void display();";
44         somDestruct: public,override,
45               cxxdecl="virtual ~GenericString();";
46         somDefaultConstVAssign: public,override;
47         somDefaultCopyInit: public,override;

```

Continued

```

48     somDefaultAssign: public, override;
49     somDefaultVAssign: public, override;
50     declarationorder = "length, data, somDefaultInit,
51         somDefaultConstCopyInit, setInit, set,
52         somDefaultConstAssign, clear, display,
53         somDestruct, somDefaultConstVAssign";
54     releaseorder:
55         _get_length,
56         s__p0,
57         _get_data,
58         s__p1,
59         setInit,
60         set,
61         clear,
62         display,
63         length,
64         data;
65     callstyle = idl;
66     dtsclass;
67     directinitclasses = "SOMObject";
68     cxxmap = "GenericString";
69     cxxdecl = "class GenericString : public virtual SOMObject";
70     dllname = genstr;
71 };
72 #endif
73 );
74 #endif /* __genstr_idl */

```

Inheriting from a C++ Class

The final example takes the previous example a step further, defining an OO COBOL class `CblString` that inherits from the DTS C++ class `GenericString`. In the class definition file, the class `CblString` inherits from `GenericString` at line 2, and defines a new method, `Identify`, at lines 10 through 13. The COBOL client program creates an instance of this class at line 20, then invokes inherited methods from `GenericString` against this instance at lines 24 and 27. At line 33, the newly introduced method `Identify` is invoked.

Definition of OO COBOL Class CblString (string/cblinh/str.cbl):

```

1     Identification Division.
2     Class-Id. CblString inherits GenericString.
3     Environment Division.
4     Configuration Section.
5     Repository.
6         Class GenericString is "GenericString".
7     Procedure Division.
8     Identification Division.
9
10    Method-Id. "identify".
11    Procedure Division.

```



```

12         Display "COBOL class CblString".
13     End Method "identify".
14
15     End Class CblString.
16

```

Client of OO COBOL Class CblString (string/cblinh/usestr.cbl):

```

1     Identification Division.
2     Program-id. "Client".
3     Environment Division.
4     Configuration section.
5     Repository.
6         Class CblString.
7     Data Division.
8     Working-storage section.
9     01 env usage pointer.
10    01 str1 usage object reference CblString.
11    01 objp usage object reference CblString.
12    01 txt pic X(25) value "string1 from COBOL".
13    01 len pic s9(9) comp.
14    01 tntp usage pointer.
15    Linkage section.
16    01 txt2 pic X(25).
17    Procedure Division.
18        Call "somGetGlobalEnvironment" returning env.
19        Display "Calling somNew".
20        Invoke CblString "somNew" returning str1
21        move length of txt to len.
22        Set tntp to address of txt.
23        Display "Calling Set".
24        Invoke str1 "Set" using by value env
25            by value tntp by value len returning objp.
26        Display "Calling _get_data".
27        Invoke str1 "_get_data" using
28            by value env returning tntp.
29        Set Address of txt2 to tntp.
30        Display txt2.
31
32        Display "Calling identify".
33        Invoke str1 "identify".
34
35        Display "Calling somFree".
36        Invoke str1 "somFree".
37        Goback.

```


The SOMObjects Object Services

This chapter provides a brief introduction to the SOMObjects 3.0 Object Services, which implement the *OMG CORBAServices: Common Object Services Specification*. Since the intent of this book is to describe aspects of SOM that are specific to DirectToSOM C++, the Object Services will not be discussed in great detail. But, in keeping with a major goal of the book, which is to provide working examples written using DirectToSOM C++, explanations and examples are provided of the more commonly used services.

Overview

The SOM Object Services consist of the following services:

- | | |
|---------------------------|--|
| Naming Service | Allows objects to be named so that they can more easily be retrieved and shared by different processes or across the network. |
| Life Cycle Service | Provides classes that support the creation and deletion of local or remote objects without needing information about their location. |

Object Identity Service	Allows a program to determine whether two objects are identical.
Externalization Service	Provides support for reading and writing objects to flat buffers, called streams, allowing objects to be copied, moved, and so on without breaking the object encapsulation.
Persistence Service	Provides support for saving an object's state persistently, so that an object can extend beyond the life of the process in which it was created.
Concurrency Service	Provides a mechanism for controlling shared access to resources through read and write locks.
Transaction Service	Allows a program to manipulate objects within a transaction, similar to the support provided by database management systems; but transactions can span across objects distributed across the network.
Event Service	Provides a means of communicating between distributed objects in a decoupled form.
Security Service	Provides authentication support to prevent clients from accessing objects unless they are authorized.

All of these except the Security Service are implementations of the *OMG CORBAServices: Common Object Services Specification* (OMG Document Number 95-3-31). (The Security Service is not OMG-compliant because the specification was approved only in January 1996.) Most of the OMG services are defined as abstract classes, from which concrete SOM implementation classes are derived. The OMG-defined classes and types start with the name **Cor**, while the corresponding SOM types start with **som** or **Som**.

This chapter provides an overview of the basic Object Services concepts, followed by explanations and examples that form the building blocks for the rest of the services, specifically, the Naming, Life Cycle, Object Identity, Externalization, and Persistent Object Services.

The information in this chapter is based on the SOMObject 3.0 beta for OS/2, which is subject to change. In addition, an internal version of the VisualAge C++ product was used to create the examples, specifically, one that supports the **SOMModule** pragma, which is used heavily by the Object Services. In order to re-create the examples, you will need to install the version of the VisualAge C++ product that follows version 3.0.

Object Life Cycle Model

The Object Services introduce a new flavor of object, called a *managed object*, is one that inherits from the Object Services class `somOS::ServiceBase` in `<somos.hh>`. The Object Services manipulate managed objects, rather than simple SOM or remote DSOM objects. Managed objects can be created either locally or remotely, but apart from some of the distinctions described here, are manipulated by client programs in much the same way as SOM or DSOM objects.

The life cycle model for an object dictates how that object is created, managed, and destroyed. Standard SOM objects are created, managed, and destroyed in `DirectToSOM C++` in much the same way as standard `C++` objects. An object is defined either automatically, statically, or dynamically, which results in storage for that object being allocated and initialized in the client process. The object is manipulated directly, using standard `C++` operations. A SOM object is transient, in that the storage for the object is deallocated before or upon program termination, either through an explicit **delete** operation, or implicitly when the scope or program in which that object was declared terminates. SOM objects are local to the current process, and cannot be manipulated through other processes.

A DSOM object is created, implicitly or explicitly, through a factory object. The storage for such objects is allocated in a remote server process, and a local proxy is used by the client process to manipulate the object. DSOM objects are deallocated in the client process through either the `SOMObject::somFree` or the `SOMObject::release` methods. Certain restrictions apply to how DSOM objects are manipulated, a major consideration being that instance data is not local to the current process. In addition, the remote object could potentially be shared by another process, which means that the client must take into account that another process may be dependent upon the object, or may update or destroy that object. DSOM objects are also transient, and do not persist beyond the existence of the server process in which they were created. If a client has a proxy for a remote object whose server has terminated, that proxy is invalid.

Managed objects can be created locally as a standard SOM object, but typically they are created remotely through a factory object, as with DSOM objects. One of the major advantages between a managed object and a SOM or DSOM object is that the state of a managed object can endure beyond the life of the process in which it was created. The possibility for persistence introduces additional considerations in handling managed objects, beyond those of shared remote objects, which will be discussed in the next section.

Managed Objects

Managed objects have three distinct components, that while highly interrelated in the makeup of the object, are also quite independent in their state:

- ♦ object reference
- ♦ in-memory object instance
- ♦ persistent state of object

The object reference is the information contained in an object proxy that is used by the server process to map from an incoming object proxy to the actual remote object. The object reference allows that object to be shared by processes other than the process in which the object was created. For DSOM objects, the object reference is always transient, meaning that the reference is valid only as long as the server process that created the reference is active. If the server process in which a DSOM object was created is terminated, any proxy references to that object are no longer valid. Remote managed objects may have either transient or persistent references. Even if the server process terminates, a persistent reference will allow that object to be re-created in memory if the server process is reactivated. This is handled by a special object in the server process called the **Object Services Server**, which will be discussed in more detail shortly.

The in-memory instance of an object refers to the state of the object that is currently available to be operated on by the application. For SOM and DSOM objects, the in-memory object instance is simply the object after it has been created and before it has been destroyed. In other words, the object itself persists only as long as its in-memory instance. For nonpersistent managed objects, the same applies.

For persistent managed objects, the in-memory instance is also created when the object is initially created, as shown in Figure 10.1. However, persistent objects also have a persistent state that endures beyond the life of the

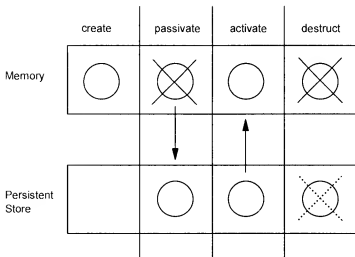


FIGURE 10.1 Managed object life cycle.

in-memory instance. The in-memory instance may be removed from memory, but saved in a persistent store. This is called *passivation* of the object. Likewise, the in-memory instance may be re-created from the persistent store, which is known as *activation* of the object. Destruction of a persistent object refers to deallocating the in-memory instance in such a way that it cannot be reactivated, and may optionally include deleting its persistent state.

As you can see, each element of a managed object is interrelated in defining the life cycle of the underlying object. However, they are not interdependent. For example, an object may have a persistent reference without persistent state, in which case the object itself can be re-created from the reference, rather than the reference simply being invalid; but its original state will be lost. An object need not have a reference at all, persistent or transient, in which case that object cannot be accessed and shared in other processes.

To distinguish the context within which an object is being created and destroyed, the **somOS::ServiceBase** class introduces six new methods for handling the creation and destruction of managed objects:

init_for_object_creation	Performs initialization when constructing a given managed object. For example, in this method, you would register the object in any frameworks or containers to which it should belong and allocate any heap or persistent storage required by the object.
uninit_for_object_destruction	Performs cleanup when the in-memory object and the object's persistent reference are being destroyed. For example, you would deregister the object from any frameworks or containers to which it belongs and deallocate any heap or persistent storage allocated by the object.
init_for_object_activation	Performs initialization when reconstructing a new in-memory instance of a previously created managed object. For example, you would allocate any heap storage required by the object.
uninit_for_object_passivation	Performs cleanup of a managed object whose in-memory instance is being deleted; but the state will be saved persistently. For example, you would deallocate any heap storage allocated by the object.
init_for_object_copy	Performs initialization of a managed object that is being copied from another service process. This method is currently not called by SOMObjects 3.0.

uninit_for_object_move

Performs cleanup of a managed object that will be moved to another server process. This method is currently not called by SOMObject 3.0.

These methods are called by the various Object Services frameworks when manipulating managed objects. There is no mapping between the C++ constructors and destructors and these methods as there is for **somDefaultInit** and **somDestruct**, so the methods must be invoked explicitly. (Note: The last two, **init_for_object_copy** and **uninit_for_object_move**, are currently not used in the SOMObjects 3.0 product.) Depending upon the needs of your class implementation, you can choose to override these methods to handle the specific type of creation or destruction being performed.

Object Services Server

An Object Services Server is a server implementation that maintains persistent references for managed objects. Each Object Services Server contains a special object, called the *Server object*, that manages persistent references.

Recall from Chapter 8, that each server process contains a SOM Object Adapter object that acts as the interface between the server program and the DSOM run time. The SOM Object Adapter creates a Server object that is assigned to the global variable **SOMD_ServerObject**, using the implementation repository to determine the class of the Server object. One of the purposes of the Server object is to create and resolve object references. The Object Adapter invokes the **SOMDServer** methods **somdSOMObjFromRef** and **somdRefFromSOMObj** against the **SOMD_ServerObject** to map between object references and the objects themselves. (Refer back to Figure 8.8.)

The **SOMDServer** object keeps an in-memory table and uses information in the proxy reference to look up the associated object; however, this table is not persistent. If the server process is terminated, the mapping information is lost. Hence, a proxy object for a remote object whose server process has terminated is invalid—there is no information to map the proxy to the remote object, even if the remote object were re-created.

In order to support persistent references, Object Services processes use a subclass of **SOMDServer**, **somOS::Server**, to create and resolve object references. This Object Services Server object maintains a persistent database that includes the mapping information from the object reference to the actual object, and information about the object, called *metadata*. The metadata consists of sufficient information to allow the Server object to re-create the object from a reference, should the original object have been destroyed due to the server process terminating.

By default, managed object references are transient, but they can be made persistent. The concept of persistent references will be discussed in more detail later in this chapter.

The default server program for DSOM server processes is **somdsvr**. For Object Services processes, a different server program is used: **somossvr**. To register a server implementation that uses the Object Services, you must indicate both the server program and the server object class. For example, to register the server implementation **IdentityServer** to support the class **Identity**, which uses the Object Services, the following would be used:

```
regimpl -A -i IdentityServer -p "somossvr.exe" \
-v "somOS::Server"
regimpl -a -i IdentityServer -c Identity
```

The **-p** option, which was used in Chapter 8 to register a user-defined server program implementation, indicates that the server implementation process should execute the program **somossvr**. The **-v** option indicates that the Object Adapter should create a Server object of type **somOS::Server**. Figure 10.2 shows the server process model for the above server implementation.

After an Object Services Server implementation is created, it must be initialized before it can be used. Among other things, initialization creates the persistent database used by the server to save persistent references. A server can be initialized by starting the server explicitly with the **-i** option, as follows:

```
somossvr -i -a IdentityServer
```

The other option is to initialize the server programmatically, using the **somos_init_persist_dbs** function. If the server implementation is not initialized prior to being used, the server process will terminate with an error. Deleting the server implementation from the implementation repository will remove any corresponding persistent database information, and the

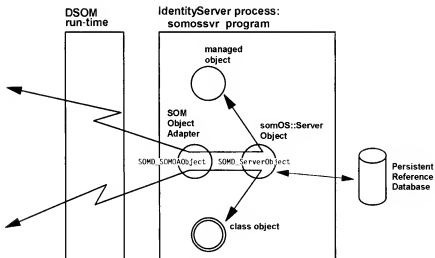


FIGURE 10.2 Object Services Server.

server would have to be reinitialized to be usable. In addition, reinitializing an existing server implementation will erase any existing persistent information for that server.

The remainder of this chapter will provide explanations and examples of several of the Object Services, specifically, the Naming, Life Cycle, Object Identity, Externalization, and Persistent Object Services. I chose these because they are the building blocks for the remainder of the services.

Naming Service

The Naming Service provides support to assign a hierarchical name to an object, known as *binding* an object to that name, and then later retrieve that object through the given name. SOM extends the OMG-defined interfaces, which only support assigning a name, with support for associating properties to that binding and performing constraint-based searches. I will consider only the basic OMG-compliant naming support in this discussion. See the SOMObjects documentation for examples using properties. The Naming Service is used quite frequently in many situations, so I will provide several examples of its use.

The central concept in the Naming Service is that of a *naming context*, which is very much like a directory in a standard file system. A naming context is also an object, and thus can be bound into another naming context in much the same way that a directory can be contained in another directory. An object is bound to a name in a specific context, which allows names to be grouped and their associated objects retrieved together, again in the same way as files in a directory can be managed.

Recall from Chapter 8 that in order to use DSOM, you must first configure the Naming Service. Part of this configuration creates a default local root naming context, to which objects and other contexts may be bound. In addition, there is a single global naming context that is distributed across all systems in the workgroup, which is bound into each local root with the name “.”.

In order to use the Naming Service, you must first retrieve a reference to the local root context. This is done by invoking the **resolve_initial_references** method against the **SOMD_ORBObject**. Recall from Chapter 8 that this method was used to obtain addressability to the DSOM Factory Service, using the lookup string “FactoryService”. In this case, the lookup string “NameService” is used, which returns a proxy that provides access to the local naming context root. Then, you can use the **bind** and **resolve** methods against this naming context to register and find objects in the Naming Service.

Registering a Name with the Naming Service

The following example illustrates the process of registering and retrieving a name in the Naming Service. The program is essentially the same as that in

Chapter 8, but uses the Naming Service instead of saving the DSOM stringified proxy to a file. At line 3 in the client program, the header file **somnm.hh** is included, which includes definitions for using the Naming Service. At lines 14 through 16, the variable **rootNC** is set to the result of resolving to the root naming context. Next, at lines 19 through 24, the IDL **sequence** variable name, of type **CosNaming::Name**, is set to indicate the name with which the object will be registered. A **CosNaming::Name** bound name is a sequence consisting of one or more **CosNaming::NameComponent** elements, where the first element is the actual name to which the object is bound, and any additional names represent context names. In this example, I am using a simple name, "MyObjidObject", so the sequence length is 1. In addition, a name can have an associated **kind** field, which is used to attach semantics to the name, instead of using the name itself to represent the object type. For example, **kind** could be set to *employee* to indicate that this is an employee object. The Naming Service does not interpret either the **name** or the **kind** field.

The program then invokes the **resolve** method against the root-naming context at line 26 to find the object in the Naming Service if it exists, assigning the result to **remoteObj**. If an error did not occur, indicating that the Naming Service found an object, that object is checked for validity. If the object is no longer valid, it is unbound from the naming context at line 30, using the name created earlier as a reference point, and **remoteObj** is set to null. Note that at line 31, an error is checked for and ignored. This is to perform error cleanup in case the object was not actually found in the Naming Service at line 26. Attempting an **unbind** in this situation would result in an error being generated, because the name was not found in the Naming Service, so there would be exception information to cleanup. We don't actually care if an error occurred on the **unbind** operation in this example.

If **remoteObj** is null at line 35, then either it was not found in the naming context, or it was found to be invalid. In either case, a new object is created at line 37 and bound to the naming context at line 39, using the **bind** method against the root-naming context, supplying the name created earlier.

At lines 43 through 46, the program prompts for a number, which is displayed at the process in which the remote object was created. The program can be run multiple times, from different processes, and as long as the server containing the remote object does not terminate, the same remote object will be found through the naming service by each program invocation.

Note that the program deallocates only the proxies for the remote object and the root-naming context. This is to allow these objects to be used again. If the remote object itself were deallocated, each program would find the object in the Naming Service, but determine it to be invalid and re-create it.

The Naming Service does not require managed objects, therefore the program uses a standard DSOM server, defined at lines 24 and 25 in the makefile.

Definition of Class Objid (naming\objid\objid.hh):

```

1  #include <som.hh>
2
3  #pragma SOMNOMangling(on)
4
5  class Objid : public SOMObject {
6      #pragma SOMClassName(*, "Objid")
7      #pragma SOMIDLPass(*, "Implementation-End", \
8          'dllname = \'objid.dll\'');
9      public:
10         void sayHello(long);
11 };

```

Implementation of Class Objid (naming\objid\objid.cpp):

```

1  #include <fstream.h>
2  #include "objid.hh"
3
4  void Objid::sayHello(long number)
5  {
6      cout << "Hello with number " << number <<
7          " with object " << this << endl;
8  }

```

Client of Class Objid (naming\objid\tstobjid.cpp):

```

1  #include <iostream.h>
2  #include <assert.h>
3  #include <somm.h>
4  #include <somd.h>
5  #include "objid.hh"
6  #include "check.h"
7
8  int main(int argc, char *argv[])
9  {
10     __SOMEnv = SOM_CreateLocalEnvironment();
11     SOMD_Init(__SOMEnv);
12
13     // resolve to root naming context
14     ExtendedNaming::ExtendedNamingContext *rootNC =
15         (ExtendedNaming::ExtendedNamingContext *)
16         SOMD_ORBObject->
17         resolve_initial_references("NameService");
18
19     CosNaming::Name name;
20     CosNaming::NameComponent nameComponent;
21     name._length = name._maximum = 1;
22     name._buffer = &nameComponent;
23     name._buffer[0].id = "MyObjidObject";
24     name._buffer[0].kind = NULL;
25

```

```

26  Objid *remoteObj = (Objid*)rootNC->resolve(&name);
27
28  if (checkError(__SOMEnv) ||
29      ! isValidRemoteObject(remoteObj)) {
30      rootNC->unbind(&name);
31      checkError(__SOMEnv); // eat unbind error
32      remoteObj = NULL;
33  }
34
35  if (! remoteObj) {
36      remoteObj = (Objid *)
37          somdCreate(__SOMEnv, "Objid", TRUE);
38      assert(! checkError(__SOMEnv, TRUE));
39      rootNC->bind(&name, remoteObj);
40      assert(! checkError(__SOMEnv, TRUE));
41  }
42
43  cout << "Enter a number: ";
44  long number;
45  cin >> number;
46  remoteObj->sayHello(number);
47  assert(! checkError(__SOMEnv, TRUE));
48
49  // deallocate proxies only
50  ((SOMDObject *)remoteObj)->release();
51  ((SOMDObject *)rootNC)->release();
52
53  SOM_DestroyLocalEnvironment(__SOMEnv);
54
55  return(0);
56 }

```

Makefile (naming\objid\makefile):

```

1  all: objid.dll tstobjid.exe somdimpl.dat
2
3  ICCOPTS = -I ..\..\include
4
5  objid.dll: objid.cpp objid.hh objidi.c
6      gcc $(ICCOPTS) /Ti+ /Ge- /B"/NOE" \
7          objid.cpp objidi.c objid.def
8      implib objid.lib objid.dll
9
10  tstobjid.exe: tstobjid.cpp
11      gcc $(ICCOPTS) /Ti+ /B"/NOE" tstobjid.cpp \
12          objid.lib ..\..\bin\sommem1.obj
13
14  objid.idl: objid.hh
15      gcc $(ICCOPTS) objid.hh
16      sc -sir -u objid.idl
17
18  objidi.c: objid.idl
19      sc -simod objid.idl

```

Continued

```

20     sc -sdef objid.idl
21
22     somdimpl.dat:
23         regimpl -D -i ObjidServer
24         regimpl -A -i ObjidServer
25         regimpl -a -i ObjidServer -c Objid

```

Creating a New Naming Context

The previous example bound an object with the name "MyObjidObject" to the root naming context, but for most applications, you will want to create subcontexts in which to register names. The following example shows how to do this. There are two aspects; the first is registering the subcontext object itself, and the second is the registering the objects within that context.

The example shows how to create a new context and bind two objects to that new context. The initial part of the program is similar to the previous example, except that we are binding a naming context, not just a name. The root naming context is resolved at lines 18 through 20, and assigned to `rootNC` as in the previous example. At lines 23 through 28, a **CosNaming::Name** sequence is created, but this time for the name of the new context, "MyNamingContext". At line 30, the root context is searched for the name, and the result is assigned to the variable `myNC`. If an error occurs or the naming context is not a valid proxy, the name is unbound from the root naming context at line 35. Note that naming contexts are also objects—they are not just names in the context tree.

If the variable `myNC` is null at line 39, then a new naming context is created and bound to the root naming context at line 40, using the **bind_new_context** method, which creates a new context object in the server process containing the target context (in this case, the server containing the root naming context corresponding to `rootNC`) and binds the name into the target context. You can also create contexts separately in two steps, using the **new_context** method, which creates a new context in the same server as the target context, and **bind_context**, which binds the context into the naming tree.

Once the new context is created, it is used to create two new objects, at lines 44 and 47, using the function `createObject`. This function accepts a target naming context and a name, and will return the object corresponding to that name in the given context, creating a new one if necessary.

The `createObject` function is shown at line 61. This function creates a **CosNaming::Name** variable at lines 64 through 69, using the `objName` parameter. The resulting variable, `name`, is used to resolve to the remote object within the naming context given by the `nc` parameter. If the remote object is found, but is not valid, the function `UnbindAll` is called, which will unbind all objects in the given naming context, the assumption being that, because both objects in the program are created at the same server, if one is no longer valid, the other will be invalid too. If `remoteObj` is null at line 81,

a new one is created at line 85 and bound into the given naming context at line 87; `remoteObj` is then returned to the caller at line 90.

The `UnbindAll` function, at line 93, illustrates the use of the `CosNaming::NamingContext::list` method, which is used at line 97 to return a list of bound objects in a given naming context. The `list` method accepts three parameters, the first being the number of bound objects that should be retrieved, the second a sequence of type `CosNaming::BindingList`, and the third an iterator of type `CosNaming::BindingIterator`. The second parameter is used to return a list of bound objects, up to the maximum given by the first parameter. The iterator is used to retrieve any subsequent objects. In this example, I have specified `INT_MAX` for the number of objects to retrieve, so the iterator will not be used. At lines 98 through 103, the function unbinds each name in the returned sequence and displays the name being unbound. Then, at line 104, the returned storage for the sequence is deallocated.

A word of warning: Be careful that you do not call `UnbindAll` with the root naming context. This will erase everything in the naming tree that was configured by `som_cfg` and you will need to reconfigure the Naming Service. I learned this the hard way, of course.

Creating a Naming Context (`naming/context/tstnc.cpp`):

```

1  #include <iostream.h>
2  #include <assert.h>
3  #include <limits.h>
4  #include <somnm.hh>
5  #include <somd.hh>
6  #include "hello.hh"
7  #include "check.h"
8
9  void UnbindAll(CosNaming::NamingContext *);
10 Hello *createObject(CosNaming::NamingContext *, char *);
11
12 int main(int argc, char *argv[])
13 {
14     __SOMEnv = SOM_CreateLocalEnvironment();
15     SOMD_Init(__SOMEnv);
16
17     // resolve to root naming context
18     ExtendedNaming::ExtendedNamingContext *rootNC =
19         (ExtendedNaming::ExtendedNamingContext *)
20         SOMD_ORBObject->resolve_initial_references(
21             "NameService");
22
23     CosNaming::Name name;
24     CosNaming::NameComponent nameComponent;
25     name._length = name._maximum = 1;
26     name._buffer = &nameComponent;
27     name._buffer[0].id = "MyNamingContext";

```

Continued

```

28     name._buffer[0].kind = NULL;
29
30     CosNaming::NamingContext *myNC =
31         (CosNaming::NamingContext *)
32         rootNC->resolve(&name);
33     if (checkError(__SOMEnv) ||
34         ! isValidRemoteObject(myNC)) {
35         rootNC->unbind(&name);
36         checkError(__SOMEnv);
37         myNC = NULL;
38     }
39     if (! myNC) {
40         myNC = rootNC->bind_new_context(&name);
41         assert(! checkError(__SOMEnv, TRUE));
42     }
43
44     Hello *remoteObj1 =
45         createObject(myNC, "MyHelloObject1");
46     remoteObj1->sayHello();
47     Hello *remoteObj2 =
48         createObject(myNC, "MyHelloObject2");
49     remoteObj2->sayHello();
50
51     ((SOMDObject *)remoteObj1)->release();
52     ((SOMDObject *)remoteObj2)->release();
53     ((SOMDObject *)myNC)->release();
54     ((SOMDObject *)rootNC)->release();
55
56     SOM_DestroyLocalEnvironment(__SOMEnv);
57
58     return(0);
59 }
60
61 Hello *createObject
62     (CosNaming::NamingContext *nc, char *objName)
63 {
64     CosNaming::Name name;
65     CosNaming::NameComponent nameComponent;
66     name._length = name._maximum = 1;
67     name._buffer = &nameComponent;
68     name._buffer[0].id = objName;
69     name._buffer[0].kind = NULL;
70
71     Hello *remoteObj = (Hello*)nc->resolve(&name);
72
73     if (checkError(__SOMEnv))
74         remoteObj = NULL;
75     else if (remoteObj &&
76         ! isValidRemoteObject(remoteObj)) {
77         UnbindAll(nc);
78         remoteObj = NULL;
79     }
80
81     if (! remoteObj) {

```



```

82         checkError(__SOMEnv);
83         cout << "Creating " << objName << endl;
84         remoteObj = (Hello *)
85             somdCreate(__SOMEnv, "Hello", TRUE);
86         assert(! checkError(__SOMEnv, TRUE));
87         nc->bind(&name, remoteObj);
88         assert(! checkError(__SOMEnv, TRUE));
89     }
90     return remoteObj;
91 }
92
93 void UnbindAll(CosNaming::NamingContext *nc)
94 {
95     CosNaming::BindingList list = {0, 0, NULL};
96     CosNaming::BindingIterator *iterator;
97     nc->list(INT_MAX, &list, &iterator);
98     for (int i = 0; i < list._length; i++) {
99         nc->unbind(&list._buffer[i].binding_name);
100         cout << "Unbinding " <<
101             list._buffer[i].binding_name._buffer[0].id
102             << endl;
103     }
104     delete list._buffer;
105 }

```

Using the Naming Service with the Message Queue Program

As promised in Chapter 8, in the next example, I updated the Message Queue program to use the Naming Service to share remote objects across processes, instead of saving the DSOM proxy to a file. I included only the relevant changes here, which are in the `lmqmgr.h` and `lmqmgr.cpp` files. The rest of the program remains unchanged.

The program now creates a naming context that is used to register and resolve the message queue server and each message queue. The only change to the `LocalMessageQueue` class definition is the addition of a new instance data member, `mqueueNC`, which will be used to hold the message queue naming context proxy, and two member functions `resolveMessageQueueContext` and `resolveMessageQueueManager`, which will resolve to the message queue context and `MessageQueueManager` objects respectively.

The constructor for the class, at line 13 in file `lmqmgr.cpp`, has been updated to initialize the DSOM environment and invokes the `resolveMessageQueueContext` and `resolveMessageQueueManager` to handle the rest of the initialization. The destructor, at line 29, simply releases the local proxies for the message queue context and `MessageQueueManager` object.

The `GetMessageQueue(int)` method, line 37, is the same as before—it simply retrieves the message queue directly from the `MessageQueueManager` object. The `GetMessageQueue(char *)` method, line 48, has been updated to

look for the message queue in the Naming Service, instead of a local file as in the example in Chapter 8. At lines 51 through 56, a **CosNaming::Name** variable is created using the message queue name supplied in the `qName` parameter to the method. At line 60, the message queue naming context given by `mqueueNC` is searched for this name. If the object is not found, the message queue is retrieved from the server and bound to the `mqueueNC` naming context at lines 67 through 69. The resulting `MessageQueue` proxy is then returned to the caller at line 71.

The `resolveMessageQueueContext` method appears at line 75. It is called by the `LocalMessageQueueManager` constructor to initialize the `mqueueNC` data member. The root naming context is resolved to at line 79. Then, at line 84 through 89, a **CosNaming::Name** variable is initialized to search for the name "MessageQueueContext". If the name is found in the naming context and is valid, then no further action is required. If the name is found, but the proxy is no longer valid, the name is unbound from the naming context at line 97. A new context is created at line 102 if a valid existing one is not found.

The `resolveMessageQueueManager` method, at line 108, is called by the `LocalMessageQueueManager` constructor to initialize the `mqmgr` data member. This method first looks for the object named "MessageQueueManager" in the `mqueueNC` naming context. If a valid object is found, no further action is necessary. If the object is found, but is no longer a valid proxy, then all names in the current context are unbound at lines 131 through 137. The assumption here is that, since the message queue manager is created by the same server process as each message queue, if the message queue manager is no longer valid, all the message queues will be invalid too. If a valid message queue manager object was not found, a new one is created and bound to the naming context at lines 141 through 166.

Class Definition of LocalMessageQueueManager (naming/mqueue/lmqmgr.h):

```

1  #ifndef LMQSERVER_H
2  #define LMQSERVER_H
3
4  #include <sommm.hh>
5
6  #include "mqmgr.hh"
7
8  class LocalMessageQueueManager {
9      MessageQueueManager *mqmgr;
10     CosNaming::NamingContext *mqueueNC;
11
12     void resolveMessageQueueContext();
13     void resolveMessageQueueManager();
14 public:
15     LocalMessageQueueManager();
16     ~LocalMessageQueueManager();
17     virtual MessageQueue *GetMessageQueue(char *);

```

```

18     virtual MessageQueue *GetMessageQueue(int);
19 };
20
21 #endif

```

**Class Implementation of LocalMessageQueueManager
(naming\mqqueue\lmqmgr.cpp):**

```

1  #include <iostream.h>
2  #include <stdio.h>
3  #include <limits.h>
4  #include <assert.h>
5
6  #include "lmqmgr.h"
7  #include "check.h"
8
9  #pragma SOMNODataDirect(on)
10 #include "mqmgr.hh"
11 #pragma SOMNODataDirect(off)
12
13 LocalMessageQueueManager::LocalMessageQueueManager()
14 {
15     // initialize DSOM
16     static Environment env;
17     SOM_InitEnvironment(&__SOMEnv = &env);
18     SOMD_Init(&__SOMEnv);
19     // Handle memory dealloc with delete/SOMFree
20     SOMD_NoORBFree();
21
22     // resolve to naming context
23     resolveMessageQueueContext();
24     // resolve to MessageQueueManager
25     resolveMessageQueueManager();
26 }
27
28 // release proxies
29 LocalMessageQueueManager::~LocalMessageQueueManager()
30 {
31     ((SOMDObject *)mqmgr)->release();
32     ((SOMDObject *)mqqueueNC)->release();
33 }
34
35 // returns the proxy for the remote
36 // MessageQueue by queue number
37 MessageQueue* LocalMessageQueueManager::
38     GetMessageQueue(int qnum)
39 {
40     return mqmgr->GetMessageQueue(qnum);
41 }
42
43
44 // returns the proxy for the remote MessageQueue
45 // by queue name queues are stored locally by name,
46 // so a local search is performed first for the queue

```

Continued

```

47 // before checking the server.
48 MessageQueue* LocalMessageQueueManager::
49     GetMessageQueue(char *qName)
50 {
51     CosNaming::Name name;
52     CosNaming::NameComponent namingComponent;
53     name._length = name._maximum = 1;
54     name._buffer = &namingComponent;
55     name._buffer[0].id = qName;
56     name._buffer[0].kind = NULL;
57
58     // look for the message queue locally first
59     MessageQueue *remoteObj =
60         (MessageQueue *)mqueueNC->resolve(&name);
61     if (checkError(__SOMEnv))
62         remoteObj = NULL;
63     if (! remoteObj) {
64         // file/object doesn't exist, so get
65         // object from server, and save
66         // the id for the object in the file
67         remoteObj = mcmgr->GetMessageQueue(qName);
68         if (remoteObj)
69             mqueueNC->bind(&name, remoteObj);
70     }
71     return remoteObj;
72 }
73
74
75 void LocalMessageQueueManager::
76     resolveMessageQueueContext()
77 {
78     // resolve to root naming context
79     ExtendedNaming::ExtendedNamingContext *rootNC =
80         (ExtendedNaming::ExtendedNamingContext *)
81         SOMD_ORBObject->
82         resolve_initial_references("NameService");
83
84     CosNaming::Name name;
85     CosNaming::NameComponent namingComponent;
86     name._length = name._maximum = 1;
87     name._buffer = &namingComponent;
88     name._buffer[0].id = "MessageQueueContext";
89     name._buffer[0].kind = NULL;
90
91     mqueueNC = (CosNaming::NamingContext *)
92         rootNC->resolve(&name);
93     if (checkError(__SOMEnv))
94         mqueueNC = NULL;
95     else if (mqueueNC &&
96             ! isValidRemoteObject(mqueueNC)) {
97         rootNC->unbind(&name);
98         checkError(__SOMEnv);
99         mqueueNC = NULL;
100     }
101     if (! mqueueNC) {

```

```

102     mqueueNC = rootNC->bind_new_context(&name);
103     assert(! checkError(__SOMEnv, TRUE));
104 }
105 ((SOMDObject *)rootNC)->release();
106 }
107
108 void LocalMessageQueueManager::resolveMessageQueueManager()
109 {
110     CosNaming::Name name;
111     CosNaming::NameComponent namingComponent;
112     name._length = name._maximum = 1;
113     name._buffer = &namingComponent;
114     name._buffer[0].id = "MessageQueueManager";
115     name._buffer[0].kind = NULL;
116
117     // look for the manager in the naming service
118     mqmgr = (MessageQueueManager *)
119         mqueueNC->resolve(&name);
120     if (checkError(__SOMEnv)) {
121         mqmgr = NULL;
122     } else if (mqmgr &&
123                ! isValidRemoteObject(mqmgr)) {
124         mqmgr = NULL;
125         // unbind all in this context, since
126         // target object is bad
127         CosNaming::BindingList list = {0, 0, NULL};
128         CosNaming::BindingIterator *iterator;
129         mqueueNC->list(INT_MAX, &list, &iterator);
130         boolean more = (iterator != NULL);
131         for (int i = 0; i < list._length; i++) {
132             mqueueNC->
133                 unbind(&list._buffer[i].binding_name);
134             cout << "Unbinding " <<
135                 list._buffer[i].binding_name._buffer[0].id
136                 << endl;
137         }
138         delete list._buffer;
139     }
140
141     if (! mqmgr ) {
142         // Get Naming Service factory service
143         // for locating factories
144         ExtendedNaming::ExtendedNamingContext *enc =
145             (ExtendedNaming::ExtendedNamingContext *)
146             SOMD_ORBObject->
147                 resolve_initial_references("FactoryService");
148         assert(enc && ! checkError(__SOMEnv, TRUE));
149
150         // Find factory (class object) for
151         // class MessageQueueManager
152         SOMClass *factory = (SOMClass *)
153             enc->find_any("class == 'MessageQueueManager'"
154                          " and alias == 'MQServer'", 0);

```

Continued

```

155         assert(factory && ! checkError(__SOMEnv, TRUE));
156
157         // release Factory Service proxy
158         ({SOMDObject *}enc)->release();
159
160         // Create a remote object through the factory
161         mqmgr = (MessageQueueManager *)factory->somNew();
162         assert(mqmgr && ! checkError(__SOMEnv, TRUE));
163
164         mqueueNC->bind(&name, mqmgr);
165         assert(! checkError(__SOMEnv, TRUE));
166     }
167 }

```

Life Cycle Services

The Life Cycle Services provide a variety of interfaces for locating and creating factory objects through which managed objects can be created. These interfaces are:

```

somLifeCycle::GenericFactory
somLifeCycle::FactoryFinder
somLifeCycle::LifeCycleObject
somLifeCycle::GenericFactoryAbstraction
somLifeCycle::Location
somLifeCycle::ServerSetLocation
somLifeCycle::ConstraintBuilder
somLifeCycle::FactoryFilter

```

The first three, **somLifeCycle::GenericFactory**, **somLifeCycle::FactoryFinder**, and **somLifeCycle::LifeCycleObject** are all implementations of OMG-compliant interfaces. The remainder are SOM extensions provided for improved usability.

The **GenericFactory** class uses an associated **FactoryFinder** object to create objects, which can be on different servers and of different class types. It is similar in concept to the DSOM **somdCreate** function. The **GenericFactoryAbstraction** is an abstract class that provides the interface used by **GenericFactory** (which derives from **GenericFactoryAbstraction**). This class can be used to provide a user-defined **GenericFactory** class.

The **FactoryFinder** class supports methods to locate factory objects subject to various constraints. The **Location**, **ServerSetLocation**, **ConstraintBuilder**, and **FactoryFilter** classes can be used to control how factories are located by a **FactoryFinder** object. The **LifeCycleObject** class can be inherited from in order to provide compliance with OMG standards.

Objects of class **GenericFactory**, **FactoryFinder**, and **ServerSetLocation** maintain persistent state, and must therefore be created in an

Objects Services Server. Programs that use instances of these classes must therefore interact with an Object Services Server process. To illustrate this process, the next example shows our familiar `Hello` class, which has been renamed to `HelloOS` and now inherits from the class `somOS::ServiceBase`, making it a managed object class.

The `HelloOS` class overrides the default constructor and destructor for the class, and the `init_for_object_creation` and `uninit_for_object_destruction` methods. As part of the initialization and destruction for managed objects, the methods `init_for_object_creation` and `uninit_for_object_destruction` must be called in order to interact with the Object Services Server object. These methods do not have implicit support for calling their parent methods, as do the default constructor and destructor, so you must call the parent methods explicitly. Even if the class does no initialization or destruction, the `init_for_object_creation` and `uninit_for_object_destruction` methods must still be overridden in order to ensure that all base classes are properly initialized and uninitialized in order for the Object Services Server to manage the objects.

In the class implementation for `HelloOS`, the default constructor and destructor (which map to `somDefaultInit` and `somDestruct`) calls these life cycle methods. This allows objects to be created through the more standard mechanisms used with DSOM. Note that most of the examples in the SOMObjects documentation create the object first with no initialization (via `somNewNoInit`), and then separately invoke the `init_for_object_creation` method, which requires releasing the interim proxy created in the first step. This is not necessary if you invoke `init_for_object_creation` from the constructor—you can simply create the object in one step.

The class implementation contains a static definition for the variable `ev`, which is assigned to `__SOMEnv` at line 4. This is required because the `somOS::ServiceBase` class is not an `oidl` class like `SOMObject`, and therefore the `init_for_object_creation` and `uninit_for_object_destruction` methods expect, and check for, a valid `Environment` parameter to be passed. The `DirectToSOM` compiler will pass `__SOMEnv` implicitly, but it must first be initialized, otherwise an exception may occur. In previous examples, this initialization was not necessary in the class implementation, because the implementation did not invoke any SOM methods that expected an `Environment` parameter to be passed. All of the `SOMObject` methods use the globally defined `Environment` variable (accessed through the `somGetGlobalEnvironment` function). As mentioned earlier, if the compiler version you are using already initializes `__SOMEnv`, this explicit initialization is not necessary.

At line 13 in the client program, the function `findFactory` is called, passing the name of the `HelloOS` class. This will return a `SOMClass` factory proxy, which is used at line 15 to create a new `HelloOS` object. The method `sayHello` is invoked on the resulting object at line 18, then the proxy and remote object are deallocated at line 20, and the factory proxy is released at line 22.

The `findFactory` function, where most of the work in this example is performed, is contained in the file `osutil.h`. (You would typically put the function in a separately compiled file, but I wanted to keep the build process for the examples as simple as possible, so I put it in a header file that is included into the example.) This file includes `somlc.hh` and `somlcdef.hh`, which contain definitions for the Life Cycle Services.

At line 35, `findFactory` calls the function `getFactoryFinder`. If the function returns a **FactoryFinder**, a **CosLifecycle::Key** variable is created to locate the desired class, at lines 40 through 43. Specifying **KIND_OBJ_INF** for the `kind` member indicates that a factory should be returned that is capable of creating objects of the class given by the `id` member. There are several other values that can be specified here—see the `SOMObjects` documentation for details. Using the information that was registered in the implementation repository at line 26 in the `makefile`, the `find_factory` method invocation at line 44 should return the `HelloOS` `SOM` class object. If the factory is found, the proxy objects for the naming context and the factory finder are released, and the factory is returned to the caller.

The `getFactoryFinder` method, at line 9 first resolves to the Naming Service at line 13. Then, a **CosNaming::Name** variable is created at lines 17 through 21, with the name of the **LifeCycleFactoryFinder** object in the global naming context (`./`). This is a default **FactoryFinder** object that is registered with the Naming Service when `DSOM` is configured, which is resolved to at line 23. The proxy for the Naming Service is released at line 26, and the **FactoryFinder** is returned at line 28.

Definition of Class `HelloOS` (`lifecyc\hello.hh`):

```

1  #include <somos.hh>
2
3  class HelloOS : public somOS::ServiceBase {
4      #pragma SOMClassName(*, "HelloOS")
5      #pragma SOMIDLPass(*, 'Implementation-End', \
6          "dllname = \"hello.dll\";")
7  public:
8      HelloOS();
9      ~HelloOS();
10     SOMObject *init_for_object_creation();
11     void uninit_for_object_destruction();
12     void sayHello();
13 };

```

Implementation of Class `HelloOS` (`lifecyc\hello.cpp`):

```

1  #include <fstream.h>
2  #include "hello.hh"
3
4  static Environment ev =
5      (SOM_InitEnvironment(&__SOMEnv = &ev), ev);

```



```

6
7 HelloOS::HelloOS()
8 {
9     init_for_object_creation();
10 }
11
12 SOMObject *HelloOS::init_for_object_creation()
13 {
14     somOS::ServiceBase::init_for_object_creation();
15     return this;
16 }
17
18 HelloOS::~HelloOS()
19 {
20     uninit_for_object_destruction();
21 }
22
23 void HelloOS::uninit_for_object_destruction()
24 {
25     somOS::ServiceBase::
26         uninit_for_object_destruction();
27 }
28
29 void HelloOS::sayHello()
30 {
31     cout << "Hello world" << endl;
32 }

```

Client of Class HelloOS (lifecyc\lsthello.cpp):

```

1 #include <iostream.h>
2 #include <assert.h>
3 #include "check.h"
4 #include "osutil.h"
5 #include "hello.hh"
6
7 int main(int argc, char *argv[])
8 {
9     __SOMEnv = SOM_CreateLocalEnvironment();
10
11     SOMD_Init(__SOMEnv);
12
13     SOMClass *factory = findFactory("HelloOS");
14
15     HelloOS *remoteObj =
16         (HelloOS *)factory->somNew();
17
18     remoteObj->sayHello();
19
20     remoteObj->somFree();
21
22     ((SOMDObject *)factory)->release();
23
24     SOMD_Uninit(__SOMEnv);

```

Continued

```

25     SOM_DestroyLocalEnvironment(__SOMEnv);
26
27     return(0);
28 }

```

Utility Functions (include\osutil.h):

```

1  #ifndef OSUTIL_H
2  #define OSUTIL_H
3
4  #include <somd.hh>
5  #include <somlc.hh>
6  #include <somlcdef.hh>
7  #include "check.h"
8
9  somLifeCycle::FactoryFinder* getFactoryFinder()
10 {
11     ExtendedNaming::ExtendedNamingContext *fenc =
12         (ExtendedNaming::ExtendedNamingContext *)
13         SOMD_ORBObject->resolve_initial_references("NameService");
14     assert(! checkError(__SOMEnv, TRUE));
15
16     // resolve to default factory finder
17     CosNaming::NameComponent nn[2] = {
18         { ".", ""},
19         { "LifeCycleFactoryFinder", ""}
20     };
21     CosNaming::Name name = {2, 2, nn};
22     somLifeCycle::FactoryFinder* ff =
23         (somLifeCycle::FactoryFinder*)fenc->resolve(&name);
24     assert(! checkError(__SOMEnv, TRUE));
25
26     ((SOMDObject *)fenc)->release();
27
28     return ff;
29 }
30
31 // finds a factory for the named class
32 SOMClass *findFactory(char *className)
33 {
34     somLifeCycle::FactoryFinder* ff =
35         getFactoryFinder();
36     if (!ff)
37         return NULL;
38
39     // locate factory proxy
40     CosNaming::NameComponent element;
41     CosLifeCycle::Key key = {1, 1, &element};
42     key._buffer[0].kind = KIND_OBJ_INF;
43     key._buffer[0].id = className;
44     SOMClass *factory = (SOMClass *)ff->find_factory(&key);
45     assert(! checkError(__SOMEnv, TRUE));
46
47     ((SOMDObject *)ff)->release();
48

```

```

49     return factory;
50 }
51
52 #endif

```

Makefile (lifecyc\makefile):

```

1  all: hello.dll tsthello.exe somdimpl.dat
2
3  ICCOPTS = /I ..\..\include
4
5  hello.dll: hello.cpp hello.hh hello.i.c
6      icc $(ICCOPTS) /Ti+ /Ge- /B*/NOE* hello.cpp \
7          hello.i.c hello.def ..\..\bin\sommeml.obj
8      implib hello.lib hello.dll
9
10 tsthello.exe: tsthello.cpp hello.hh
11     icc $(ICCOPTS) /Ti+ /B*/NOE* tsthello.cpp \
12         hello.lib ..\..\bin\sommeml.obj
13
14 hello.idl: hello.hh
15     icc $(ICCOPTS) hello.hh
16     sc -sir -u hellc.idl
17
18 hello.i.c: hello.idl
19     sc -simod hello.idl
20     sc -sdef hello.idl
21
22 somdimpl.dat:
23     regimpl -D -i HelloOSServer
24     regimpl -A -i HelloOSServer \
25         -p "somasvr.exe" -v "somOS::Server"
26     regimpl -a -i HelloOSServer -c HelloOS

```

Object Identity Service

The Object Identity Service allows a program to determine whether two objects are identical; that is, whether they refer to the same instance, rather than simply having identical state. When dealing with remote objects, it is difficult, if not impossible, to determine if two different proxies refer to the same remote object. The Object Identity Service provides a mechanism for doing so through the **somOS::ServiceBase** class. This class provides a unique identity for each object instance, along with an **is identical** method that can be used to test for identity. An instance of a class that inherits from **somOS::ServiceBase** is an *identifiable* object.

The **is identical** method is guaranteed to produce an accurate result. However, because it involves a remote method invocation, it can be expensive to call it. For example, when searching through a large set of objects to determine identity, a remote method call would be made for each test. Con-

sequently, for efficiency reasons, each identifiable object also contains a randomly generated numerical attribute: **constant_random_id**. The attribute value is generated based on the time of the object creation, and is constant for the life of the object. While it is not guaranteed to be unique across all objects, it can be used as an efficient first test for object identity. For example, each time an object is added to a collection, that object's **constant_random_id** value can be saved locally. When a search is performed, the locally stored **constant_random_id** value can be checked against that of the search object. Only if the **constant_random_id** attribute values match is a call to **is_identical** necessary, to find out if the objects really match.

As an example of using the **is_identical** method, the following program creates a remote instance of the class *Identity*, which derives from **somOS::ServiceBase** in file **somos.hh**, using the **somdCreate** function at line 14. At line 17, the method **returnSelf** is invoked against the proxy, which returns a new proxy object (the **returnSelf** method simply returns the **this** pointer, which is marshaled into a new object proxy by DSOM on the return from the method). The local address for each proxy is different, but they both represent the same remote object. At line 19, the local address of each proxy is printed, followed by the result of calling the **is_identical** method at line 22. The output for this program is:

```
remoteObj1: 0xd9bd10, remoteObj2: 0xd9d210
Objects are identical: true
```

Definition of Class Identity (identity\identity.hh):

```
1  #include <somos.hh>
2
3  class Identity : public somOS::ServiceBase {
4      #pragma SOMClassName(*, "Identity")
5      #pragma SOMIDLPass(*, "Implementation-End", \
6          "dllname = \"identity.dll\";")
7  public:
8      Identity();
9      ~Identity();
10     SOMObject *init_for_object_creation();
11     void uninit_for_object_destruction();
12     Identity *returnSelf();
13 };
```

Implementation of Class Identity (identity\identity.cpp):

```
1  #include <iostream.h>
2  #include "identity.hh"
3
4  static Environment ev =
5      {SOM_InitEnvironment(__SOMEnv = &ev), ev};
6
```

```

7 Identity::Identity()
8 {
9     init_for_object_creation();
10 }
11
12 SOMObject *Identity::init_for_object_creation()
13 {
14     somOS::ServiceBase::init_for_object_creation();
15     return this;
16 }
17
18 Identity::~Identity()
19 {
20     uninit_for_object_destruction();
21 }
22
23 void Identity::uninit_for_object_destruction()
24 {
25     somOS::ServiceBase::uninit_for_object_destruction();
26 }
27
28 Identity *Identity::returnSelf()
29 {
30     return this;
31 }

```

Client of Class Identity (identity\tstid.cpp):

```

1 #include <iostream.h>
2 #include <assert.h>
3 #include <somd.hh>
4 #include "check.h"
5 #include "identity.hh"
6
7 int main(int argc, char *argv[])
8 {
9     __SOMEnv = SOM_CreateLocalEnvironment();
10
11     SOMD_Init(__SOMEnv);
12
13     Identity *remoteObj1 = (Identity *)
14         somdCreate(__SOMEnv, "Identity", TRUE);
15
16     // get a second proxy for same remote object
17     Identity *remoteObj2 = remoteObj1->returnSelf();
18
19     cout << "remoteObj1: " << (void *)remoteObj1 <<
20         ", remoteObj2: " << (void *)remoteObj2 << endl;
21
22     cout << "Objects are identical: " <<
23         (remoteObj1->is_identical(remoteObj2) ?
24         "true" : "false");
25

```

Continued

```

26     ((SOMDObject *)remoteObj2)->release();
27     remoteObj1->somFree();
28
29     SOMD_Uninit(__SOMEnv);
30     SOM_DestroyLocalEnvironment(__SOMEnv);
31
32     return(0);
33 }

```

Makefile (identity\makefile):

```

1  all: identity.dll tstid.exe somdimpl.dat
2
3  ICCOPTS = /I ../include
4
5  identity.dll: identity.cpp identity.hh identiti.c
6      gcc $(ICCOPTS) /Ti+ /B- /B*/NOE* identity.cpp \
7      identiti.c identity.def ../bin/sommeml.obj
8      implib identity.lib identity.dll
9
10 tstid.exe: tstid.cpp identity.hh
11     gcc $(ICCOPTS) /Ti+ /B*/NOE* tstid.cpp \
12     identity.lib ../bin/sommeml.obj
13
14 identity.idl: identity.hh
15     gcc $(ICCOPTS) identity.hh
16     sc -sir -u identity.idl
17
18 identiti.c: identity.idl
19     sc -simod identity.idl
20     sc -sdef identity.idl
21
22 somdimpl.dat:
23     regimpl -D -i IdentityServer
24     regimpl -A -i IdentityServer \
25     -p "somosvr.exe" -v "somOS::Server"
26     regimpl -a -i IdentityServer -c Identity

```

Externalization Service

The Externalization Service provides a framework for writing and reading an object's state to and from a *stream*. This is known as *externalizing* or *internalizing* the object respectively. Objects that can be externalized or internalized through a stream are called *streamable* objects. Externalizing a streamable object writes a representation of that object's state to a stream; internalizing reads that representation and defines the state of the target streamable object as defined by that representation. Once an object's state is externalized, it can be internalized into multiple target objects. The Externalization Service allows an object to be moved or copied without breaking

the compromising object encapsulation. The underlying storage medium for the stream is transparent to the class implementation, allowing the same streamable object to be written to different types of streams.

Streamable objects must be instances of a class that inherits from the Externalization Service class **somStream::Streamable**. This class inherits from both **somOS::ServiceBase** and the OMG-defined class **CosStream::Streamable**. Two methods from each of these base classes must be overridden by any class that derives from **somStream::Streamable**: the methods **externalize_to_stream** and **internalize_from_stream** are inherited from **CosStream::Streamable**, while **init_for_object_creation** and **uninit_for_object_destruction** are inherited from **somOS::ServiceBase**. When a client program indicates that a streamable object should be externalized, the method **externalize_to_stream** transfers the object's state to the stream. When the client indicates that the state should be internalized, the **internalize_from_stream** method transfers the object's state from the stream into the target object.

As shown in Figure 10.3, a stream is defined by two classes that reference one another: **somExternalization::Stream** (*streams*) and **somStream::StreamIO** (*streamIOs*). Instances of **somExternalization::Stream** are responsible for the semantics of stream usage, while **somStream::StreamIO** instances are responsible for the representation of object state. Two classes are used in the implementation so that the stream semantics and representation can be varied independently. To ensure that both parts of a stream object are initialized correctly, streams must be created through a **somExternalization::StreamFactory** object, which is supplied the class names that implement each part of the stream to be created. Up to this point, the term stream has been used generically to refer to the joint object described previously. From now on, the term stream will mean an instance of either **somExternalization::Stream** or **somExternalization::OSStream**.

Streams are implemented as an instance of either **somExternalization::Stream** or **somExternalization::OSStream**. These implementations differ

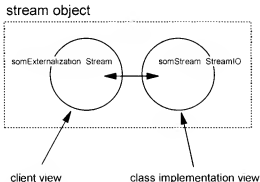


FIGURE 10.3 Stream object representation.

mainly in how references to embedded objects are handled, which will be discussed later in this section.

*StreamIO*s are instances of **somStream::StreamIO**, and are implemented by one of the following classes: **somStream::StandardStreamIO**, **somStream::StringStreamIO**, or **somStream::MemoryStreamIO**. These representations differ mainly in how data is stored in the *streamIO* buffer, and are discussed in more detail in the *SOMObjects* documentation. Class implementations of **externalize_to_stream** and **internalize_from_stream** are passed a pointer to a **CosStream::StreamIO** object, the abstract OMG-defined class from which **somStream::StreamIO** derives.

As an example of the Externalization Service support, the following program illustrates the process of creating and using a stream. The class `Employee` inherits from the class **somStream::Streamable**, which is included by the file **somestrn.hh**, and declares three data members: `name`, `department`, and `salary`. It declares a default constructor and destructor and a method `dump` to display an object's state. In addition, the class supplies definitions for the methods **init_for_object_creation**, **uninit_for_object_destruction**, **externalize_to_stream**, and **internalize_from_stream**.

In the class implementation, the default constructor, line 7, invokes the **init_for_object_creation** method, line 12, which first invokes **somStream::Streamable::init_for_object_creation** (this method in turn invokes **somOS::ServiceBase::init_for_object_creation**). Then the data members are initialized, and the object is returned.

The **externalize_to_stream** method is shown at line 41. It accepts a single parameter, which is the target stream for the object. Each data member is written to the stream using a **CosStream::StreamIO** method appropriate to the type. Then, the parent **externalize_to_stream** method is invoked at line 47 to allow the parent class to write any instance data. The owned instance data and parent instance data can be written in any order, but it must be the same as that used to read the data back from the stream, as illustrated in the **internalize_from_stream** method at line 51.

The **internalize_from_stream** method accepts two parameters, a **CosStream::StreamIO** and a **FactoryFinder**. The **FactoryFinder** is used for internalizing embedded object references, and is discussed in more detail following this example. Each data member is read from the stream in the same order in which it was written, taking care to delete any preexisting dynamically allocated storage. The parent **internalize_from_stream** method is then called at line 61, again in the same order with respect to the other data members written to the stream.

The client program shows how to externalize and internalize an object to and from a stream. Because the `Employee` class is a managed object, it must be created through an Object Services process. However, I wanted to illustrate the Externalization Service as simply as possible, without creating and manipulating remote objects. To achieve this, I created an in-process Object Services server thread, which shares the same address space as the

client thread. The server thread is created by calling the `startServerThread` function at line 13. This function is contained in the `osthread.h` file, and will be discussed subsequently.

At line 15, a local `Employee` object is dynamically allocated. This object is then assigned some data values, and displayed to standard output at line. The next step is to externalize this object to a stream. At line 24 a `somExternalization::StreamFactory` object is created, which is used at lines 27 through 31 to create a stream object with `somExternalization::OSSStream` semantics and a `somStream::StringStreamIO` representation. The return type is assigned to a `somExternalization::OSSStream` object, because that is the view of the stream used by the client.

If the stream is created successfully, the `externalize` method is invoked against it at line 34, passing the `Employee` object created earlier. This method will in turn invoke the `Employee::externalize_to_stream` method, passing the stream as a parameter, which will write the object data to the stream.

At line 38, a new `Employee` object is created by invoking the `internalize` method against the stream. The `internalize` method accepts an optional `FactoryFinder` method, which it will use to create the new object. If the `FactoryFinder` is null, the Externalization Service will locate the default `cosLifeCycle::FactoryFinder` from the global naming context, as shown in the Life Cycle Service example earlier. A new object will be created through the `FactoryFinder`, and initialized by invoking `init_for_object_creation` against the object. Then, the `internalize_from_stream` method will be invoked against the object to read the stream data into the object.

If the `internalize` method completes successfully, the object data is printed to standard output at line 41. This will produce an identical result as the previous output operation at line 20.

The `startServerThread` appears at line 26 in the file `osthread.h`. The implementation is essentially the same as that of the user-defined server program in Chapter 8, with the addition of the call to `somos_init_services` at line 37 and the thread creation at lines 41 through 45. A thread is created that calls the `threadProc` defined at line 14, which starts the event processing loop for the server. In order to use the in-thread server process, the server implementation must be registered as multithread-capable, which is done with the `-m on` option in the makefile at line 18.

Definition of Class Employee (extern\employee.hh):

```

1  #include <somestrm.hh>
2
3  class Employee : public somStream::Streamable {
4      #pragma SOMClassName(*, "Employee")
5      #pragma SOMIDLPass(*, "Implementation-End", \
6          "dllname = \"employee.dll\";");
7      public:
```

Continued

```

8      char *name;
9      short department;
10     float salary;
11
12     Employee();
13     ~Employee();
14     void dump();
15
16     SOMObject *init_for_object_creation();
17     void uninit_for_object_destruction();
18     void externalize_to_stream(CosStream::StreamIO *);
19     void internalize_from_stream(
20         CosStream::StreamIO *,
21         CosLifecycle::FactoryFinder *);
22 };

```

Implementation of Class Employee (extern\employee.cpp):

```

1  #include <iostream.h>
2  #include "employee.hh"
3
4  static Environment ev =
5      (SOM_InitEnvironment(__SOMEnv = &ev), ev);
6
7  Employee::Employee()
8  {
9      init_for_object_creation();
10 }
11
12 SOMObject *Employee::init_for_object_creation()
13 {
14     somStream::Streamable::
15         init_for_object_creation();
16     name = NULL;
17     salary = department = 0;
18     return this;
19 }
20
21 Employee::~Employee()
22 {
23     uninit_for_object_destruction();
24 }
25
26 void Employee::uninit_for_object_destruction()
27 {
28     if (name)
29         delete name;
30     somStream::Streamable::
31         uninit_for_object_destruction();
32 }
33
34 void Employee::dump()
35 {
36     cout << "Name: " << name << ", department "

```

```

37         << department << ", salary: $"
38         << salary << endl;
39     }
40
41     void Employee::externalize_to_stream(
42         CosStream::StreamIO *stream)
43     {
44         stream->write_string(name);
45         stream->write_short(department);
46         stream->write_float(salary);
47         somStream::Streamable::
48             externalize_to_stream(stream);
49     }
50
51     void Employee::internalize_from_stream(
52         CosStream::StreamIO *stream,
53         CosLifecycle::FactoryFinder *ff)
54     {
55         if (name)
56             delete name;
57         name = stream->read_string();
58         department = stream->read_short();
59         salary = stream->read_float();
60         somStream::Streamable::
61             internalize_from_stream(stream, ff);
62     }
63

```

Client of Class Employee (extern\stemp.cpp):

```

1  #include <assert.h>
2  #include <somd.hh>
3  #include <somosutl.hh>
4
5  #include "check.h"
6  #include "osthread.h"
7  #include "employee.hh"
8
9  int main(int argc, char *argv[])
10 {
11     __SOMEnv = SOM_CreateLocalEnvironment();
12     SOMD_Init(__SOMEnv);
13     startServerThread("EmployeeServer");
14
15     Employee *emp = new Employee;
16
17     emp->name = "Arthur Dent";
18     emp->department = 321;
19     emp->salary = 200.34;
20     emp->dump();
21
22     // need a StreamFactory to create an OSStream
23     somExternalization::StreamFactory *factory =
24         new somExternalization::StreamFactory;

```

Continued

```

25     assert (! checkError(__SOMEnv, TRUE));
26
27     somExternalization::OSStream *stream =
28         (somExternalization::OSStream *)
29         factory->create_with_types(
30             "somExternalization::OSStream",
31             "somStream::StringStreamIO");
32     assert (! checkError(__SOMEnv, TRUE));
33
34     stream->externalize(emp);
35     assert(! checkError(__SOMEnv, TRUE));
36
37     Employee *emp2 =
38         (Employee *)stream->internalize(NULL);
39     assert(! checkError(__SOMEnv, TRUE));
40
41     emp2->dump();
42
43     delete factory;
44     delete stream;
45     delete emp;
46     delete emp2;
47     SOM_DestroyLocalEnvironment(__SOMEnv);
48 }

```

Thread Utility Functions (include/osthread.h):

```

1  #include <assert.h>
2  #include <somd.hh>
3  #include <somosutl.hh>
4
5  #define INCL_DOSPROCESS
6  // Support server thread creation
7  #include <os2.h>
8
9  #include "check.h"
10
11 // The entry point for this routine is passed as
12 // an argument to startSvrThread() below. It
13 // starts the event loop of the server.
14 ULONG threadProc()
15 {
16     Environment *__SOMEnv = somGetGlobalEnvironment();
17
18     SOMD_SOMOAObject->execute_request_loop(SOMD_WAIT);
19     checkError(__SOMEnv, TRUE);
20     return(NULL);
21 }
22
23
24 // Starts a Object Service thread in the current process
25 // for handling managed objects locally
26 void startServerThread(char *server)

```

```

27 {
28     TID taskID;
29     Environment *__SOMEnv = somGetGlobalEnvironment();
30     const int stack_size = 65536;
31
32     SOMD_SOMOAObject = new SOMOA;
33     SOMD_ImplDefObject =
34         SOMD_ImplRepObject->find_impldef_by_alias(server);
35     checkError(__SOMEnv, TRUE);
36
37     somos_init_services(TRUE);
38     SOMD_SOMOAObject->impl_is_ready(SOMD_ImplDefObject);
39     somos_init_services_afterimpl(TRUE);
40
41     if (DosCreateThread(&taskID, (PFNTHREAD)threadProc,
42         0, 0, stack_size) != 0) {
43         somPrintf("\n%s failed/n", server);
44         exit(1);
45     }
46 }

```

Makefile (extern\makefile):

```

1  all: ttemp.exe employee.idl somdimpl.dat
2
3  ICCOPTS = /I ../..//include
4
5  # no dll to enable OS Server thread
6  ttemp.exe: employee.hh employee.cpp ttemp.cpp
7      icc $(ICCOPTS) /Ti+ /B*/NOE* ttemp.cpp \
8      employee.cpp ..\..\bin\sommem1.obj
9
10 employee.idl: employee.hh
11     icc $(ICCOPTS) employee.hh
12     sc -sir -u employee.idl
13     sc -sdef employee.idl
14
15 somdimpl.dat:
16     regimpl -D -i EmployeeServer
17     regimpl -A -i EmployeeServer -p "somosvr.exe" \
18         -v "somOS::Server" -m on
19     regimpl -a -i EmployeeServer -c Employee

```

Embedded Object References

For classes that contain references to other SOM objects, there are several methods that are used to write and read such data members. **write_object_value** is used when the containing class owns the object; in other words, it is responsible for creating and deleting that object. This method will always write the object value to the stream. **write_object** is used when the class only references the object and does not own it. This method may

or may not write the object to the stream, depending upon the stream implementation.

For reading objects from a stream, there is just a single method, **read_object**, which accepts a **FactoryFinder** and **CosStream::Streamable** parameter:

```
::CosStream::Streamable* read_object
(
    ::CosLifeCycle::FactoryFinder*,
    ::CosStream::Streamable*);
```

If the second parameter is null, the Externalization Service uses the **FactoryFinder** to create a new in-memory instance of the next object in the stream. This is why the **FactoryFinder** parameter is passed to the **internalize_from_stream** method—so that it can be used to internalize any embedded object references.

The second parameter can be used to prevent a given object from being re-created multiple times in memory. If the second parameter is not null, depending upon how the object was originally written to the stream, if the object supplied as the second parameter is identical to the next object to be read, that object will be skipped in the stream. See the SOMObjects documentation for details.

Recursive Save Operations

To avoid recursive object saves, the Externalization Service compares an object to be written against each existing object in the stream. If the objects are identical, the object will not be written. This is achieved using the Identity Service **is_identical** method. For efficiency, the Externalization Service also stores the **constant_random_id** attribute for each object and uses that as a first test for object identity.

Diamond Top Class Hierarchy

While the Externalization Services can detect a recursive save operation on a particular object, it does not detect when a given part of an object has already been written (or read). Because each class is responsible for invoking the parent **externalize_from_stream** and **internalize_from_stream** methods, if two classes inherit from the same parent, resulting in a *diamond top* class hierarchy, it is possible that the parent may be read or written twice. In order to avoid this situation, the **somStream::StreamIO** class supports an **already_streamed** method that can be invoked against a stream to determine whether the current portion has already been read or written. This method is not necessary unless your hierarchy has a diamond top with a potential for multiple parent operations.

Persistence

There are two aspects to the persistence support provided by the Object Services: persistent references and persistent data. If an object has a persistent reference, then that reference has been stored in the Object Services Server object database. If the server terminates and is restarted, that reference will still be valid—the Server object will use its saved information to re-create an object of the appropriate type; however, the state of that object will have been lost. Allowing the state information to be saved and restored persistently is the purpose of the Persistent Object Services.

Persistent References

To make a reference persistent, the method **make_persistent_ref** can be invoked against the Object Services Server, either from within or outside the object. This instructs the Server to create an entry for that object reference in its persistent database. Or, you can have the persistence reference be created automatically by inheriting from the class **somOS::serviceBasePRef**. When objects that are derived from this class are created, they will automatically be registered with the Object Services Server when the **somOS::serviceBasePRef::init_for_object_creation** parent method is invoked.

The following provides an example of using the class **HelloOSPRef**, which inherits from **somOS::serviceBasePRef**, defined by the header **somos.hh**. The class definition and implementation are fairly straightforward. In the client program, the root naming context is searched for the object called "myObject" at line 27. If the object is not found or is found but is no longer a valid reference, the name is unbound from the naming context at line 32, and a new object is created and bound to the root naming context at lines 39 through 44.

The first time the program is run, the object will not be found in the naming context, so a new one will be created and registered. Subsequent executions of the program will find the object in the Naming Service rather than creating a new one each time. This all looks pretty normal compared to the examples we've seen so far. However, if the server process in which the remote object was created is terminated, when the client program is rerun, the object reference registered in the Naming Service will still be valid. When **isValidRemoteObject** attempts to invoke a remote request, this will create a new server process (or use the existing process if the server has already been reactivated), which will locate the supplied object reference in its database, and will re-create the object, resulting in a successful method invocation. **isValidRemoteObject** will return **TRUE**, and a new object will not be re-created by the client program. The only thing that will cause the reference to be invalid and the object to be re-created by the client is if the

server implementation is reinitialized, which would delete its database of persistent references.

Contrast this example with the Naming Service examples shown earlier. If the server process in which any of the remote objects in these earlier examples were terminated, the object references would become invalid and `isValidRemoteObject` would return `FALSE`, requiring that the objects be re-created by the client.

Definition of Class HelloOSPref (possom/persref/hello.hh):

```

1  #ifndef HELLOPR_HH
2  #define HELLOPR_HH
3  #include <somos.hh>
4
5  class HelloOSPref : public somOS::ServiceBasePref {
6      #pragma SOMClassName(*, "HelloOSPref")
7      #pragma SOMIDLPass(*, "Implementation-End", \
8          "dllname = \"hello.dll\";")
9      public:
10         HelloOSPref();
11         ~HelloOSPref();
12         SOMObject *init_for_object_creation();
13         void uninit_for_object_destruction();
14         void sayHello();
15     };
16
17 #endif

```

Implementation of Class HelloOSPref (possom/persref/hello.cpp):

```

1  #include <fstream.h>
2  #include "hello.hh"
3
4  static Environment ev =
5      (SOM_InitEnvironment(&ev, ev));
6
7  HelloOSPref::HelloOSPref()
8  {
9      cout << __FUNCTION__ << endl;
10     init_for_object_creation();
11 }
12
13 SOMObject *HelloOSPref::init_for_object_creation()
14 {
15     cout << __FUNCTION__ << endl;
16     somOS::ServiceBasePref::
17         init_for_object_creation();
18     return this;
19 }
20
21 HelloOSPref::~HelloOSPref()
22 {

```



```

23     cout << __FUNCTION__ << endl;
24     unittest_for_object_destruction();
25 }
26
27 void HelloOSPref::unittest_for_object_destruction()
28 {
29     cout << __FUNCTION__ << endl;
30     somOS::ServiceBasePref::
31         unittest_for_object_destruction();
32 }
33
34 void HelloOSPref::sayHello()
35 {
36     cout << "Hello world" << endl;
37 }

```

Client of Class HelloOSPref (possom/persref/tsthello.cpp):

```

1  #include <iostream.h>
2  #include <assert.h>
3  #include <somd.hh>
4  #include "check.h"
5  #include "hello.hh"
6
7  int main(int argc, char *argv[])
8  {
9      __SOMEnv = SOM_CreateLocalEnvironment();
10
11     SOMD_Init(__SOMEnv);
12
13     CosNaming::Name name;
14     CosNaming::NameComponent nameComponent;
15     name._length = name._maximum = 1;
16     name._buffer = &nameComponent;
17     name._buffer[0].id = "myObject";
18     name._buffer[0].kind = NULL;
19
20     // resolve to root naming context
21     ExtendedNaming::ExtendedNamingContext *rootNC =
22         (ExtendedNaming::ExtendedNamingContext *)
23         SOMD_ORBObject->
24             resolve_initial_references("NameService");
25
26     HelloOSPref *remoteObj =
27         (HelloOSPref *)rootNC->resolve(&name);
28
29     if (checkError(__SOMEnv) ||
30         ! isValidRemoteObject(remoteObj)) {
31         cout << "Unbinding" << endl;
32         rootNC->unbind(&name);
33         checkError(__SOMEnv); // eat unbind error
34         remoteObj = NULL;
35     }
36

```

Continued

```

37     if (! remoteObj) {
38         cout << "Creating object" << endl;
39         remoteObj = (HelloOSPref *)
40             somdCreate(__SOMEnv, "HelloOSPref", TRUE);
41         assert(! checkError(__SOMEnv, TRUE));
42         rootNC->bind(&name, remoteObj);
43         assert(! checkError(__SOMEnv, TRUE));
44     }
45
46     remoteObj->sayHello();
47
48     ((SOMDObject *)rootNC)->release();
49     ((SOMDObject *)remoteObj)->release();
50
51     SOMD_Uninit(__SOMEnv);
52     SOM_DestroyLocalEnvironment(__SOMEnv);
53
54     return(0);
55 }

```

Persistent Object Service (POSSOM)

So far, all the examples we have considered handled transient objects only. The discussion in the previous section applied to persistent references only. A persistent reference allows the object to be re-created with the appropriate type, but the object state is lost. With a transient object, the state of that object exists only as long as the program that created it exists. When the program ends, the state of the object is lost. There are many circumstances, however, when you would like to keep an object around after the process or program has ended. This is typically done in C++ through either class libraries that support object I/O or through handwritten code that saves and restores the contents of an object.

The Persistent Object Service (known as POSSOM) allows you to save objects so that they can exist after the process or program that created them has terminated. POSSOM provides support for transparently saving and restoring objects to a variety of datastores. The saved data can be stored in one of three IBM-supplied datastores, or you can subclass from the POSSOM Framework to manage persistence through more specialized repositories. The POSSOM model is designed so the data store implementation is independent of the class implementation. Once a class is enabled as a POSSOM class, it can be saved to any datastore without modifying the class implementation.

Defining a POSSOM Class

Unlike DSOM, where any class type can support distributed objects, you must explicitly specify classes that are capable of supporting persistent objects by deriving from certain classes. The POSSOM implementation uses the externalization service to save and restore object data. A POSSOM class

must inherit from the **somStream::Streamable** class and provide overrides of the **externalize_to_stream** and **internalize_from_stream** methods. In addition, a POSSOM class must also inherit from a POSSOM programming model class. At the time of writing, there were three programming model classes supplied with the SOMObjects 3.0 beta: explicit persistence, implicit restore persistence, and implicit persistence. The first is a CORBA-compliant implementation, and the latter two are IBM extensions. It is expected that only the explicit persistence model will be made available with the formal release of SOMObjects version 3.0, so I will be discussing the explicit persistence model, which is the class **somPersistencePO::PO**.

The **somPersistencePO::PO** class inherits from the **somOS::ServiceBasePRef**, which provides persistent reference support for instances of the class. Inheriting from this class or **somStream::Streamable** also makes the instances managed objects. Therefore, the POSSOM class must also override the managed object methods: **init_for_object_creation**, **uninit_for_object_destruction**, **init_for_object_activation**, **uninit_for_object_passivation**, **init_for_object_copy**, **uninit_for_object_move**. So far, we have used only the first two of these methods. Using POSSOM, we will see how the **init_for_object_activation** and **uninit_for_object_passivation** methods come into play. (Recall that the latter two methods are not used by SOMObjects 3.0, so I will override them, but they won't be called.)

As an example, the following shows the definition of the POSSOM class **PCount**. It inherits from **somStream::Streamable** and **somPersistencePO::PO**, and will supply overrides of methods introduced by these classes, as indicated as lines 15 through 26. The class also supplies a constructor and destructor and a single data member, **count**, which is designated as an attribute.

Definition of POSSOM Class PCount (possom\pcount\pcount.hh):

```

1  #include <somppo.hh>
2  #include <somestrm.hh>
3
4  class PCount : public somPersistencePO::PO,
5                public somStream::Streamable {
6
7      #pragma SOMClassName(*, "PCount")
8      #pragma SOMIDLPass(*, "Implementation-End", \
9          "dllname = \"pcount.dll\";")
10
11  public:
12      PCount();
13      ~PCount();
14      short count;
15      #pragma SOMAttribute(count)
16
17      SOMObject *init_for_object_creation();
18      SOMObject *init_for_object_reactivation();
19      SOMObject *init_for_object_copy();
20      void uninit_for_object_destruction();
21      void uninit_for_object_passivation();

```

Continued

```

20     void uninit_for_object_move();
21
22     void externalize_to_stream(
23         CosStream::StreamIO *);
24     void internalize_from_stream(
25         CosStream::StreamIO *,
26         CosLifeCycle::FactoryFinder *);
27 };

```

POSSOM Components

The preceding is the plumbing necessary to enable a class to be stored persistently through POSSOM. In order to actually store the object, another component is necessary: a *persistence identifier* (PID). The PID is created at run time and attached to the object to be saved or restored. It supplies information about the underlying datastore to be used for the operation and the location within that datastore for the object. The datastores are managed by *Persistent Data Service* (PDS) objects. There are three IBM-supplied PDS classes, allowing classes to be saved in a variety of datastores: a POSIX flat file, a B-Tree indexed file, or a DB2 relational database. For example, the PID for a POSIX flat file PDS supplies the name of the file in which the data should be stored. Throughout the examples, I will be using the POSIX flat file PDS, but the basic concepts are the same for all three.

The final piece of the puzzle is the *Persistent Object Manager*. This is a DSOM object that is created implicitly in the server process. Its main purpose is to route POSSOM requests against an object to the appropriate PDS.

To illustrate the interactions between the various components, Figure 10.4 illustrates how an object is stored to the persistent datastore. Using the explicit persistence model, an object is stored by invoking the method **store** against it. This method is implemented in the **somPersistencePO::PO** class, and is typically not overridden by the class implementor.

When a **store** method is invoked against an object (step 1), the implementation for the **store** method finds the Persistent Object Manager and

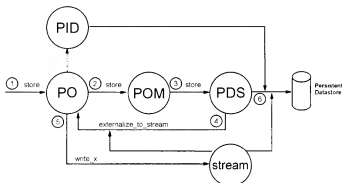


FIGURE 10.4 Saving a POSSOM object.

invokes **store** against the POM, passing the object (step 2). The POM **store** method uses the PID associated with the object to look up the Persistent Data Service for that object, and invokes **store** against the PDS, passing the object (step 3). The PDS store implementation creates a stream and invokes **externalize_to_stream** against the object, passing the stream (step 4). The **externalize_to_stream** implementation for the object writes the object data to the stream (step 5) and returns. Then, the PDS saves the stream in the appropriate datastore, using the object's PID to determine where the object should be stored (step 6).

This model allows the object implementation to be completely independent of the underlying datastore used to actually store the object. The persistent ID controls the type of the datastore where the object is saved.

A Simple Example

Now let's look at a programming example. The following shows the class `PCount`, described previously. The constructor, in the implementation file at line 7, calls **init_for_object_creation** as in earlier examples, because every time we create an object through the constructor, we are creating the initial instantiation of that object. Note that the destructor, at line 12, no longer calls **uninit_for_object_destruction**. This is because the destructor will be called both to remove the object from memory (passivate it) and to destroy it, so the destructor should only perform tasks that are related to deleting the in-memory instance of an object, and the lifecycle methods should handle any uninitialized specific to the passivation or final destruction of a managed object. I will discuss how objects are passivated later in this section.

The override of **init_for_object_creation**, line 17, simply invokes the parent methods and initializes the value of `count` to 0. The remainder of the lifecycle methods, lines 27 through 65, simply invoke the corresponding parent methods. Note that because the class has two parents, the corresponding methods for both parents must be called. The need to call both parent methods is also why we must override these methods in this example, but not previously. In previous examples, the classes had only a single parent, so the parent method would be called by default if the derived class did not supply an override. However, in this situation, we have two parent classes to initialize, which would not be handled correctly unless the methods were overridden and each parent initialized explicitly. The **externalize_to_stream** and **internalize_to_stream** implementations, lines 67 and 75, simply write and read the value of `count` to and from the passed stream.

As a very simple example of using a POSSOM class, the client program will create a `PCount` instance, and then store and restore that instance. The `PCount` instance is created in the server process at line 16. This will create an instance of `PCount` in the server process, invoke the default constructor against that object, which will invoke **init_for_object_creation**, and then will return a persistent reference for that object to the client program.

At line 20, a persistence identifier for that object is also created as a DSOM object. The persistent reference type is **somPersistencePOSIX::PID_POSIX**, indicating that the underlying datastore will be a POSIX flat file. At line 25, the **pathName** attribute of this PID is set to `"pcount.po"`, which specifies that the target datastore is the file of that name. Depending upon the desired datastore, different mechanisms are used to indicate the target datastore. For example, to create a B-Tree datastore PID, you would instantiate an object of type **somPersistenceBTREE::PID_BTREE** and set the **datastore_name** and **object_key** attributes in that PID. Refer to the SOMObjects documentation for further details. At line 28, the PID is attached to the persistent object by assigning it to the object's **p** attribute.

At line 30, the object count is set to 10. Then at line 35, the object is stored to the datastore, using the **store** method. This will follow the process described earlier of locating the POM, which in turn locates the PDS, calls **externalize_to_stream** for that object, and saves the resulting stream to the target datastore. In this example, the object will be saved in the file `pcount.po`. The **store** method accepts a single parameter, which is a PID. This can be used as a temporary override of the PID associated with the target object. Passing NULL indicates that the target object PID should be used.

At line 38, the value of `count` is set to 0. Then at line 43, the object is restored from the datastore using the **restore** method. This will follow pretty much the same process as for a **store** operation, but **internalize_to_stream** will be called to restore the object's data from the persistent datastore.

At line 50, the **Delete** method is invoked against the object, passing the PID. This method deletes the underlying datastore entry; in this case, the file `pcount.po` will be deleted. The **uninit_for_object_destruction** method is invoked at line 53, which only handles any uninitialization required by the parent classes on object destruction, as this class has no such requirements. Then, both the proxy and remote `PCount` object and PID are destroyed at lines 56 and 57.

Note that the POSSOM classes are not implicitly registered with any of the server implementations supplied with SOMObjects. Therefore, you must register both the PID and the PDS class with a server implementation, as shown in the makefile at lines 35 through 38 for the **somPersistencePOSIX::PID_POSIX** and **somPersistencePOSIX::PDS_POSIX** classes. In addition, a server process that handles POSSOM objects must be registered as multithread-capable using **-m on**, as shown at line 33 in the makefile. The output of this program is:

```
Before save count is: 10
After save count is: 0
After restore count is: 10
```

Definition of POSSOM Class PCount (possom\pcount\pcount.hh):

```
1 #include <somppo.hh>
2 #include <somestrm.hh>
```

```

3
4 class PCount : public somPersistencePO::PO,
5               public somStream::Streamable {
6     #pragma SOMClassName(*, "PCount")
7     #pragma SOMIDLPass(*, "Implementation-End", \
8       *dllname = \"pcount.dll\";*)
9     public:
10        PCount();
11        ~PCount();
12        short count;
13        #pragma SOMAttribute(count)
14
15        SOMObject *init_for_object_creation();
16        SOMObject *init_for_object_reactivation();
17        SOMObject *init_for_object_copy();
18        void uninit_for_object_destruction();
19        void uninit_for_object_passivation();
20        void uninit_for_object_move();
21
22        void externalize_to_stream(
23            CosStream::StreamIO *);
24        void internalize_from_stream(
25            CosStream::StreamIO *,
26            CosLifecycle::FactoryFinder *);
27 };

```

***Implementation of POSSOM Class PCount
(possom\pcount\pcount1.cpp):***

```

1  #include <iostream.h>
2  #include "pcount.hh"
3
4  static Environment ev =
5      (SOM_InitEnvironment(__SOMEnv = &ev), ev);
6
7  PCount::PCount()
8  {
9      init_for_object_creation();
10 }
11
12 PCount::~PCount()
13 {
14     // don't call uninit_for_object_destruction
15 }
16
17 SOMObject *PCount::init_for_object_creation()
18 {
19     somPersistencePO::PO::
20         init_for_object_creation();
21     somStream::Streamable::
22         init_for_object_creation();
23     count = 0;
24     return this;
25 }

```

Continued

```

26
27 SOMObject *PCount::init_for_object_reactivation()
28 {
29     somPersistencePO::PO::
30         init_for_object_reactivation();
31     somStream::Streamable::
32         init_for_object_reactivation();
33     return this;
34 }
35
36 SOMObject *PCount::init_for_object_copy()
37 {
38     somPersistencePO::PO::init_for_object_copy();
39     somStream::Streamable::init_for_object_copy();
40     return this;
41 }
42
43 void PCount::uninit_for_object_destruction()
44 {
45     somPersistencePO::PO::
46         uninit_for_object_destruction();
47     somStream::Streamable::
48         uninit_for_object_destruction();
49 }
50
51 void PCount::uninit_for_object_passivation()
52 {
53     somPersistencePO::PO::
54         uninit_for_object_passivation();
55     somStream::Streamable::
56         uninit_for_object_passivation();
57 }
58
59 void PCount::uninit_for_object_move()
60 {
61     somPersistencePO::PO::
62         uninit_for_object_move();
63     somStream::Streamable::
64         uninit_for_object_move();
65 }
66
67 void PCount::externalize_to_stream(
68     CosStream::StreamIO *stream)
69 {
70     stream->write_short(count);
71     somStream::Streamable::
72         externalize_to_stream(stream);
73 }
74
75 void PCount::internalize_from_stream(
76     CosStream::StreamIO * stream,
77     CosLifeCycle::FactoryFinder *ff)
78 {
79     count = stream->read_short();

```



```

80     somStream::Streamable::
81         internalize_from_stream(stream, ff);
82 }

```

Client of POSSOM Class PCount (possom\pcount\tst1.cpp):

```

1  #include <iostream.h>
2  #include <assert.h>
3  #include <somd.hh>
4  #include <sompposx.hh>
5  #include "pcount.hh"
6
7  #include "check.h"
8
9  int main(int argc, char *argv[])
10 {
11     __SOMEnv = SOM_CreateLocalEnvironment();
12     SOMD_Init(__SOMEnv);
13
14     // create the persistent object
15     PCount *pobj =
16         (PCount *)smdCreate(__SOMEnv, "PCount", TRUE);
17     assert(! checkError(__SOMEnv, TRUE));
18
19     // create a persistent id
20     somPersistencePOSIX::PID_POSIX *pid =
21         (somPersistencePOSIX::PID_POSIX *)
22         smdCreate(__SOMEnv,
23             "somPersistencePOSIX::PID_POSIX", TRUE);
24     assert(! checkError(__SOMEnv, TRUE));
25     pid->pathName = "pcount.po";
26
27     // assign persistent id to the object
28     pobj->p = pid;
29
30     pobj->count = 10;
31     cout << "Before save count is: "
32          << pobj->count << endl;
33
34     // save the object
35     pobj->store(NULL);
36     assert(! checkError(__SOMEnv, TRUE));
37
38     pobj->count = 0;
39     cout << "After save count is: "
40          << pobj->count << endl;
41
42     // restore the object
43     pobj->restore(NULL);
44     assert(! checkError(__SOMEnv, TRUE));
45
46     cout << "After restore count is: "
47          << pobj->count << endl;

```

Continued

```

48
49 // delete persistent data
50 pObj->Delete(pid);
51
52 // uninitialized object for destruction
53 pObj->uninit_for_object_destruction();
54
55 // delete object and pid
56 pObj->somFree();
57 pid->somFree();
58
59 SOM_DestroyLocalEnvironment(__SOMEnv);
60 }

```

Makefile (possom\pcount\makefile):

```

1 all: pcount.dll tst1.exe tst2.exe tst3.exe somdimpl.dat
2
3 ICCOPTS = -DEXPORT=_Export /I ..\..\include
4
5 pcount.dll: pcount.cpp pcount.hh pcounti.c
6     gcc $(ICCOPTS) /Ti+ /Ge- /B*/NOE* pcount.cpp \
7         pcounti.c pcount.def ..\..\bin\sommeml.obj
8     implib pcount.lib pcount.dll
9
10 tst1.exe: tst1.cpp pcount.hh
11     gcc &$(ICCOPTS) /Ti+ /B*/NOE* tst1.cpp pcount.lib \
12         ..\..\bin\sommeml.obj
13
14 tst2.exe: tst2.cpp pcount.hh
15     gcc &$(ICCOPTS) /Ti+ /B*/NOE* tst2.cpp pcount.lib \
16         ..\..\bin\sommeml.obj
17
18 tst3.exe: tst3.cpp pcount.hh
19     gcc &$(ICCOPTS) /Ti+ /B*/NOE* tst3.cpp pcount.lib \
20         ..\..\bin\sommeml.obj
21
22 pcount.idl: pcount.hh
23     gcc &$(ICCOPTS) pcount.hh
24     sc -sir -u pcount.idl
25
26 pcounti.c: pcount.idl
27     sc -simod pcount.idl
28     sc -sdef pcount.idl
29
30 somdimpl.dat:
31     regimpl -D -i PCountServer
32     regimpl -A -i PCountServer -p "somosvr.exe " \
33         -v "somOS::Server" -m on
34     regimpl -a -i PCountServer -c PCount
35     regimpl -a -i PCountServer -c \
36         somPersistencePOSIX::PID_POSIX
37     regimpl -a -i PCountServer -c \
38         somPersistencePOSIX::PDS_POSIX

```

Object Services Server and POSSOM

Previously, I discussed the concept of a persistent reference and noted that if the server process is terminated, a method invocation against an existing object reference would cause the server to re-create that object, rather than indicate that the object reference is invalid. The server process is able to re-create the object because it stores information about that object, called *metadata*, with the persistent reference in its database.

For POSSOM objects, one of the pieces of information that is stored with the metadata is the PID for that object. If the server process in the previous example were terminated and then restarted, when the test program attempted to access that object, a `PCount` object would be re-created in-memory in the server process. This would first invoke `init_for_object_reactivation` against the object, and then would invoke `restore` against the object, using the saved PID. This would cause the object to be restored from the persistent datastore. So, even if the server process were terminated, a subsequent method invocation using an existing persistent reference to that object would not only be valid, but the underlying data would be current too! The only way that the PID would become invalid is if the server implementation were reinitialized, which would delete its database.

The Object Services Server object is not actually responsible for storing the PID and restoring the object. This is achieved through overrides of the two `somOS::ServiceBase` methods `capture` and `reinit`. When the Object Services Server stores metadata for an object, it invokes the method `capture` against that object, which is overridden by any classes that need to save persistent metastate information. The `somPersistencePO::PO` class provides an override of `capture` that saves the PID in the metastate database. After the server re-creates an object, it invokes `init_for_object_reactivation`, and then invokes the method `reinit` against the object. `somPersistencePO::PO` provides an override of `reinit` that sets the PID for the object. The object is then restored using its PID.

When the server process terminates normally, the method `somos::Server::passivate_all_objects` is invoked against the Object Services Server object. This invokes `somos::Server::passivate_object` for each registered object, which invokes `capture` against each object, followed by `uninit_for_object_passivation`. Finally, the in-memory version of the object is deallocated, which will invoke `somDestruct`, which maps to the C++ destructor. This is another reason not to call `uninit_for_object_passivation` from the destructor. The POSSOM Framework will have already called it prior to freeing the storage, so it would be called twice if the destructor also called it.

Now that you understand a little more about how persistent objects are managed, let's look at another example. The previous example was useful in that it showed how to use POSSOM to save and restore an object. However, it was not a particularly good application of persistence, as we could have just as easily saved the value in a variable rather than store and restore it. In the following program, I updated the example to save the

PCount instance in the Naming Service, and to release only the local proxies and not destroy the object when the program terminates.

At line 17, the function `initServer` is called. This function handles a problem with server reactivation when an object has persistent data. The server process was hanging when initializing some of its data during an object reactivation. `initServer` simply creates an object in the server, saves it, and deletes it, which causes the server data to be initialized prior to reactivating the object. I was using the SOMObjects 3.0 beta; this problem should be fixed in the official release of SOMObjects 3.0, so you should not need to call `initServer`.

At line 18, the function `findPCount` is called, which will search the Naming Service for an existing object and return it, or create a new one if the object does not exist or is invalid. The program displays the count value at line 20, increments it at line 21, and then stores the object at line 24. Note that passing `pobj->p` to the `store` method is redundant; I could simply pass null instead.

The function `findPCount`, at line 34, first looks in the Naming Service for an object called "PCountObject", at line 48. If the object is found and it is still valid, it is returned; otherwise, it is unbound from the Naming Service at line 52, and a new object is created at lines 59 through 70. At line 59, an instance of PCount is created at the server. Then, at lines 64 through 70, a PID is created and assigned the file name `pcount.po`. This PID is assigned to `pobj` at line 73, and `pobj` is bound into the Naming Service at line 75.

The first time this program is run, a new PCount instance will be created and bound into the Naming Server, and the value of `count` displayed will be 0. Each subsequent time this program is run, `count` will be incremented and saved in the datastore. If the server process were terminated in between one of the program runs, the next execution would cause the server process to reactivate that object and restore its persistent data, so the count would continue to increment, using the most currently stored value of `count`.

Restoring through the Object Services Server (possom\pcount\tst2.cpp):

```

1  #include <iostream.h>
2  #include <assert.h>
3  #include <somd.hh>
4  #include <somppox.hh>
5  #include "pcount.hh"
6
7  #include "check.h"
8
9  PCount *findPCount();
10 void initServer();
11
12 int main(int argc, char *argv[])
13 {
14     __SOMEnv = SOM_CreateLocalEnvironment();
15     SOMD_Init(__SOMEnv);

```

```

16
17  initServer();
18  PCount *pobj = findPCount();
19
20  cout << "Count is: " << pobj->count << endl;
21  pobj->count++;
22
23  // save the object
24  pobj->store(pobj->p);
25  assert(! checkError(__SOMEnv, TRUE));
26
27  ((SOMDObject *)pobj->p)->release();
28  ((SOMDObject *)pobj)->release();
29
30  SOMD_Uninit(__SOMEnv);
31  SOM_DestroyLocalEnvironment(__SOMEnv);
32 }
33
34 PCount *findPCount()
35 {
36     ExtendedNaming::ExtendedNamingContext *rootNC =
37         (ExtendedNaming::ExtendedNamingContext *)
38         SOMD_ORBObject->
39             resolve__nitial_references("NameService");
40
41     CosNaming::Name name;
42     CosNaming::NameComponent nameComponent;
43     name._length = name._maximum = 1;
44     name._buffer = &nameComponent;
45     name._buffer[0].id = "PCountObject";
46     name._buffer[0].kind = NULL;
47
48     PCount *pobj = (PCount*)rootNC->resolve(&name);
49
50     if (checkError(__SOMEnv) ||
51         ! isValidRemoteObject(pobj)) {
52         rootNC->unbind(&name);
53         checkError(__SOMEnv); // eat unbind error
54         pobj = NULL;
55     }
56
57     if (! pobj) {
58         // create the persistent object
59         pobj = (PCount *)
60             somdCreate(__SOMEnv, "PCount", TRUE);
61         assert(! checkError(__SOMEnv, TRUE));
62
63         // create a persistent id
64         somPersistencePOSIX::PID_POSIX *pid =
65             (somPersistencePOSIX::PID_POSIX *)
66             somdCreate(__SOMEnv,
67                 "somPersistencePOSIX::PID_POSIX",
68                 TRUE);
69         assert(! checkError(__SOMEnv, TRUE));
70         pid->pathName = "pcount.po";

```

Continued

```

71
72     // assign persistent id to the object
73     pObj->p = pid;
74
75     rootNC->bind(&name, pObj);
76     assert(! checkError(__SOMEnv, TRUE));
77 }
78
79 ((SOMObject *)rootNC)->release();
80
81     return pObj;
82 }
83
84
85 // This function is needed only as a workaround
86 // for a problem in the server when reactivating
87 // persistent objects after the server has been
88 // restarted.
89 void initServer()
90 {
91     PCount *tmp = {PCount *}
92     somdCreate(__SOMEnv, "PCount", TRUE);
93
94     somPersistencePOSIX::PID_POSIX *pid =
95     (somPersistencePOSIX::PID_POSIX *)
96     somdCreate(__SOMEnv,
97     "somPersistencePOSIX::PID_POSIX",
98     TRUE);
99     assert(! checkError(__SOMEnv, TRUE));
100     pid->pathName = "dummy.po";
101     tmp->p = pid;
102     tmp->store(tmp->p);
103     tmp->Delete(pid);
104     tmp->uninit_for_object_destruction();
105     tmp->somFree();
106     pid->somFree();
107 }

```

Rather than save the object explicitly in the client, as is done in the previous example, you could instead save the object only when the server process is terminating. This has the advantage of reducing the store operations to a minimum. Note, however, that data may be lost if the server terminates abnormally, because **passivate_all_objects** will not be invoked.

For example, in the following I changed the implementation **uninit_for_object_passivation** for class **PCount** (file `possom\pcount\pcount2.cpp`) to invoke the **store** method as follows:

```

void PCount::uninit_for_object_passivation()
{
    store(p);
    somPersistencePO::PO::uninit_for_object_passivation();
    somStream::Streamable::uninit_for_object_passivation();
}

```

If, or when, the server process terminates, the object will be saved to the datastore, from which the object will later be reactivated when needed. The client program no longer needs to store the object at all, so the **main** function becomes (`possom\pcount\tst3.cpp`):

```
int main(int argc, char *argv[])
{
    __SOMEnv = SOM_CreateLocalEnvironment();
    SOMD_Init(__SOMEnv);

    initServer();
    PCount *pobj = findPCount();

    cout << "Count is: " << pobj->count << endl;
    pobj->count++;

    {(SOMDObject *)pobj->p}->release();
    {(SOMDObject *)pobj}->release();

    SOMD_Uninit(__SOMEnv);
    SOM_DestroyLocalEnvironment(__SOMEnv);
}
```

Persistent Message Queue

As a final example of POSSOM, I have rewritten the Message Queue application once more, this time to save the data structures persistently. This example is based on the Naming Service example shown earlier in this chapter, which is based on the version in Chapter 8, *DSOM*. This application is actually an interesting use of POSSOM, in that it demonstrates the handling of embedded object references.

The model I have chosen is to rely on the **passivate_all_objects** method to save the message queues when the server process is terminating, and to store the `MessageQueueManager` object whenever a new queue is added. I had originally planned to rely on **passivate_all_objects** to save the `MessageQueueManager` also, but it turned out there is a dependency problem. Because the `MessageQueueManager` must reference each `MessageQueue` object to save a reference to it, if any of the `MessageQueue` objects are passivated before the `MessageQueueManager`, this results in an exception when attempting to reference those objects, because passivation will remove the instance from memory. There is no fixed order in which objects are passivated, so, as I discovered, it is quite possible that a `MessageQueue` object could be passivated before the `MessageQueueManager` is.

The first step in enabling the application for POSSOM is adding **`somStream::Streamable`** and **`somPersistencePO::PO`** classes as base classes for the `MessageQueue` and `MessageQueueManager` classes, and to add overrides of the appropriate methods from each parent. This is shown in the updated header file `mqueue.hh`. Note that because these are overrides of

existing methods, they do not affect, nor need to appear in, the release order for the class. The `LocalMessageQueueClass` definition is included here, but is unchanged from the previous example using the Naming Service.

The first change to the `MessageQueue` implementation file is the initialization of `__SOMEnv` at line 5. Again, later versions of the compiler should implicitly initialize this variable for you. The `MessageQueue::Queue` class has not changed at all, either in definition or implementation. The next change is to the `MessageQueue()` default constructor, at line 23, which now calls **`init_for_object_creation`** to perform all initialization. The `MessageQueue(char *)` constructor, at line 28, also now relies on **`init_for_object_creation`** for most initialization, but still sets the queue name. The remainder of the original `MessageQueue` methods are unchanged from previous versions.

The next change is the addition of the overridden methods at lines 104 through the end of the file. At line 104 through 114, the **`init_for_object_creation`** method invokes the parent methods and initializes the object data. The call to **`somRenewNoInit`** at line 107 was necessary to work around a problem where the parent data was not being initialized properly to null. (Again, I am using a beta release; this should be fixed for the product release.) At line 116, **`init_for_object_reactivation`** invokes the parent methods and initializes the instance data. **`uninit_for_object_passivation`** stores the object in the persistent datastore. I am passing the PID to **`store`**, but it is redundant, and is the same as passing null.

The **`externalize_to_stream`** method at line 162 writes the `MessageQueue` contents to a stream. First, the name of the queue is written using **`write_string`**, followed by the number of messages, using **`write_short`**. Then, for each message in the queue, that message is written using **`write_string`**. **`internalize_from_stream`**, line 175, reads the data back into the object. First the `Clear` method is invoked to make sure that any existing storage is properly deallocated, followed by deallocating the storage for the queue name if necessary. Then, the queue name is read from the stream, followed by the queue count. Using the count as a loop guard, each message is read back and added to the queue using the `Send` method. Note that the stored queue count is used only as a loop guard, the queue count is not updated directly, as this will be handled by the `Send` method invocations.

In the `MessageQueueManager` class implementation, the constructor at line 17 has also been updated to invoke **`init_for_object_creation`** to handle any initialization. In addition, to work around another problem, I save a proxy for the default **`FactoryFinder`** in the variable `savedFF`, which will be used later at line 154.

The destructor, which previously deleted each `MessageQueue` instance, now is a no-op. This is because the `MessageQueue` objects are now managed by the Object Services Server, which will passivate and delete them implicitly. In fact, as mentioned earlier, if the `MessageQueueManager` passivation or destruction references a `MessageQueue` instance, that instance may have already been passivated and deallocated, resulting in an exception.

`GetMessageQueue(char *)` has not changed much, except for the addition of storing the `MessageQueueManager` whenever a new queue is added, at line 46. `GetMessageQueue(int)` has not changed at all.

The life cycle methods, from lines 59 through 120, are fairly self-explanatory. The **`externalize_to_stream`** method, line 128, is more interesting. First, it counts the number of active queues and writes that number to the stream. Then, for each active queue, the queue is written to the stream using the **`write_object`** method. As discussed previously, there are two methods that you can use for writing embedded object to a stream: **`write_object`** and **`write_object_value`**. For POSSOM, **`write_object`** does not write the contents of the object to the stream; rather it writes a *stringified reference* to that object, whereas **`write_object_value`** will invoke **`externalize_to_stream`** against that object and cause the object contents to be written to the stream. Apart from the difference in storage requirements, a major difference between the two is what happens when the object is read from the stream using **`read_object`**, which is used in the **`internalize_from_stream`** method at line 157. When a stringified reference is read, it causes the referenced object to be reactivated and the address of this object to be returned. However, if the object value is saved using **`write_object_value`**, **`read_object`** will simply create a new in-memory instance of that object and invoke **`internalize_from_stream`** to fill that object in.

The difference becomes apparent in this example when the client attempts to refer to a previously created `MessageQueue` instance found in the Naming Service after the server process has been terminated. If a stringified reference were written for the `MessageQueue`, when the `MessageQueueManager` was reactivated, that `MessageQueue` would also be reactivated and the persistent reference would simply operate on that previously reactivated object. If, however, the object value were written and read, the object would not have been reactivated, so invoking a method on a previously created `MessageQueue` would result in the object being reactivated, creating a new, separate in-memory instance of the object. The `MessageQueueManager` in-memory version of the object would be different from that of the reactivated `MessageQueue` object.

Because each embedded object can be referenced outside of the containing object, **`write_object_value`** is not appropriate for storing the contained objects. If the container were the only means of accessing the object, however, this would be an appropriate method to use. One other distinction between the two is that with **`write_object_value`**, the container and all its embedded objects must be saved and restored together, using the same PID (that of the container). When using **`write_object`**, the container and each object can be saved and restored separately using different PIDs.

The **`internalize_from_stream`** method, at line 147, reads the `MessageQueueManager` contents from the stream. At lines 152 through 154, I had to work around another problem with reactivation of objects. This uses the saved **`FactoryFinder`**, `savedFF`, to prevent the Externalization Service

from looking for a **FactoryFinder** object in the Naming Service, which is where the problem was occurring. This should also be fixed in the actual release of the product. Lines 155 through 158 read each object, which, being a stringified reference, will cause the objects to be reactivated.

The additions to the `LocalMessageQueue` implementation are mostly in the area of assigning persistent IDs to the created objects. At line 22, the `initServer` function is called to get around the reactivation problem mentioned in the `PCount` example. Again, this should not be necessary for the shipped product. The `GetMessageQueue(char *)` method, line 52, searches for a `MessageQueue` by name in the Naming Service first. If the queue is not found, it is created by invoking `GetMessageQueue` against the saved `mqmgr` object. A PID is assigned to the returned queue at lines 71 through 83, using the queue name with an extension of `.pds`. The only other change is to the `resolveMessageQueueManager` method at line 125, where a PID is assigned to the `MessageQueueManager` object at lines 165 through 171.

Note that, if the server process had been terminated and restarted, invoking `isValidRemoteObject` at line 140 in the `resolveMessageQueueManager` method would cause the server to restore the `MessageQueueManager` object, which in turn would result in each `MessageQueue` object also being restored because the stringified references were internalized. Thus, by the time the program starts executing the loop in the `main` function, the `MessageQueueManager` and all the `MessageQueue` objects will be active.

Definition of POSSOM Class *MessageQueue* (*possom\mqqueue\mqqueue.hh*):

```

1  #ifndef MQQUEUE_H
2  #define MQQUEUE_H
3
4  #include <somppo.hh>
5  #include <somestrm.hh>
6
7  #define SUCCESS 0
8  #define FAIL 1
9  #define MAX_QUEUE_NAME_LEN 20
10 #define MAX_MESSAGE_LEN 256
11
12 class MessageQueue : public somPersistencePO::PO,
13                     public somStream::Streamable {
14     #pragma SOMClassName(*, "MessageQueue")
15     #pragma SOMIDLPass(*, "Implementation-End", \
16         *dllname = "mqqueue.dll";)
17     #pragma SOMIDLPass(*, "Implementation-Begin", \
18         *memory_management = corba;")
19     struct Mqueue {
20         Mqueue *next;
21         char* message;
22         Mqueue(char *);
23         ~Mqueue();

```

```

24     };
25     Mqueue *mq, *last;
26     int count;
27 public:
28     char *name;
29     #pragma SOMAttribute(name, readonly)
30     #pragma SOMIDLPass(*, "Implementation-End", \
31         "_get_name : object_owns_result;")
32     MessageQueue();
33     MessageQueue(char *);
34     ~MessageQueue();
35     virtual int Count();
36     virtual void Clear();
37     virtual int Send(char *);
38     virtual int Receive(char **);
39     #pragma SOMMethodName(Receive, "Receive")
40     #pragma SOMIDLDecl(Receive, \
41         "long Receive(out string arg1)")
42     virtual void Dump();
43
44     SOMObject *init_for_object_creation();
45     SOMObject *init_for_object_reactivation();
46     SOMObject *init_for_object_copy();
47     void uninit_for_object_destruction();
48     void uninit_for_object_passivation();
49     void uninit_for_object_move();
50
51     void externalize_to_stream(CosStream::StreamIO *);
52     void internalize_from_stream(
53         CosStream::StreamIO *,
54         CosLifecycle::FactoryFinder *);
55     #pragma SOMReleaseOrder{ \
56         /* 1 */ MessageQueue(char*), \
57         /* 2 */ Clear(), \
58         /* 3 */ Send(char*), \
59         /* 4 */ Receive(char**), \
60         /* 5 */ Dump(), \
61         /* 6 */ Count(), \
62         /* 7 */ name)
63 };
64
65 #endif

```

**Definition of POSSOM Class MessageQueueManager
(*possom\mqueue\mqmgr.hh*):**

```

1  #ifndef MQSERVER_H
2  #define MQSERVER_H
3
4  #include <somppo.hh>
5  #include <somestrm.hh>
6  #include "mqueue.hh"
7

```

Continued

```

8  #define MAX_QUEUES 20
9
10 class MessageQueueManager :
11     public somPersistencePO::PO,
12     public somStream::Streamable {
13     MessageQueue *mqueues[MAX_QUEUES];
14 public:
15     #pragma SOMClassName(*, "MessageQueueManager")
16     #pragma SOMIDLPass(*, "Implementation-End", \
17         "dllname = \"mqmgr.dll\";")
18     #pragma SOMIDLPass(*, "Implementation-Begin", \
19         "memory_management = corba;")
20     MessageQueueManager();
21     ~MessageQueueManager();
22     virtual MessageQueue *GetMessageQueue(char *);
23     virtual MessageQueue *GetMessageQueue(int);
24
25     SOMObject *init_for_object_creation();
26     SOMObject *init_for_object_reactivation();
27     SOMObject *init_for_object_copy();
28     void uninit_for_object_destruction();
29     void uninit_for_object_passivation();
30     void uninit_for_object_move();
31
32     void externalize_to_stream(CosStream::StreamIO *);
33     void internalize_from_stream(
34         CosStream::StreamIO *,
35         CosLifecycle::FactoryFinder *);
36 };
37
38 #endif

```

**Definition of Class LocalMessageQueueManager
(*possom/mqueue/lmqmgr.h*):**

```

1  #ifndef LMQSERVER_H
2  #define LMQSERVER_H
3
4  #include <somnm.hh>
5
6  #include "mqmgr.hh"
7
8  class LocalMessageQueueManager {
9      MessageQueueManager *mqmgr;
10     CosNaming::NamingContext *mqueueNC;
11
12     void resolveMessageQueueContext();
13     void resolveMessageQueueManager();
14 public:
15     LocalMessageQueueManager();
16     ~LocalMessageQueueManager();
17     virtual MessageQueue *GetMessageQueue(char *);
18     virtual MessageQueue *GetMessageQueue(int);
19 };

```

```

20
21 #endif

```

**Implementation of POSSOM Class MessageQueue
(*possom\mqueue\mqueue.cpp*):**

```

1  #include <iostream.h>
2  #include <assert.h>
3  #include "mqueue.hh"
4
5  static Environment ev =
6      (SOM_InitEnvironment(&ev, ev));
7
8  MessageQueue::Mqueue(char *elemMessage)
9  {
10     next = NULL;
11     message = new char[strlen(elemMessage) + 1];
12     assert(message != NULL);
13     strcpy(message, elemMessage);
14 }
15
16 MessageQueue::Mqueue::~Mqueue()
17 {
18     delete message;
19     if (next)
20         delete next;
21 }
22
23 MessageQueue::MessageQueue()
24 {
25     init_for_object_creation();
26 }
27
28 MessageQueue::MessageQueue(char *qname)
29 {
30     init_for_object_creation();
31     name = new char[strlen(qname) + 1];
32     assert(name != NULL);
33     strcpy(name, qname);
34 }
35
36 MessageQueue::~MessageQueue()
37 {
38     Clear();
39     if (name)
40         delete name;
41 }
42
43 int MessageQueue::Send(char *message)
44 {
45     Mqueue *elem;
46     if (! (elem = new Mqueue(message)))
47         return FAIL;

```

Continued

```

48     if (mq == NULL) {
49         mq = last = elem;
50     } else {
51         last->next = elem;
52         last = elem;
53     }
54     ++count;
55     return SUCCESS;
56 }
57
58 int MessageQueue::Receive(char **bufp)
59 {
60     if (!mq) {
61         // can return NULL with DSOM 3.0
62         *bufp = NULL;
63         return FAIL;
64     }
65     Mqueue *elem = mq;
66     mq = mq->next;
67     --count;
68     if (last == elem)
69         last = NULL;
70     // DSOM will deallocate this if caller-owned
71     *bufp = new char[strlen(elem->message) + 1];
72     assert(*bufp);
73     strcpy(*bufp, elem->message);
74     // so don't delete entire chain
75     elem->next = NULL;
76     delete elem;
77     return SUCCESS;
78 }
79
80 int MessageQueue::Count()
81 {
82     return count;
83 }
84
85 void MessageQueue::Dump()
86 {
87     int i = 1;
88     cout << "Dumping queue " << name << endl;
89     for (Mqueue *cur = mq; cur != NULL;
90          cur=cur->next, i++)
91         cout << '\t' << i << ": "
92              << cur->message << endl;
93 }
94
95 void MessageQueue::Clear()
96 {
97     if (mq != NULL) {
98         delete mq;
99         mq = last = NULL;
100     }
101     count = 0;

```

```

102 }
103
104 SOMObject *MessageQueue::init_for_object_creation()
105 {
106     // workaround: make sure parent data initialized
107     __ClassObject->somRenewNoInit(this);
108     somPersistencePO::PO::init_for_object_creation();
109     somStream::Streamable::init_for_object_creation();
110     name = NULL;
111     last = mq = NULL;
112     count = 0;
113     return this;
114 }
115
116 SOMObject *MessageQueue::init_for_object_reactivation()
117 {
118     somPersistencePO::PO::
119         init_for_object_reactivation();
120     somStream::Streamable::
121         init_for_object_reactivation();
122     name = NULL;
123     last = mq = NULL;
124     count = 0;
125     return this;
126 }
127
128 SOMObject *MessageQueue::init_for_object_copy()
129 {
130     somPersistencePO::PO::init_for_object_copy();
131     somStream::Streamable::init_for_object_copy();
132     return this;
133 }
134
135 void MessageQueue::uninit_for_object_destruction()
136 {
137     somPersistencePO::PO::
138         uninit_for_object_destruction();
139     somStream::Streamable::
140         uninit_for_object_destruction();
141 }
142
143 // Is safe to call store here, because this object does
144 // not depend upon any other objects that may have been
145 // already passivated. Contrast with MessageQueueServer,
146 // which cannot be stored on passivation.
147 void MessageQueue::uninit_for_object_passivation()
148 {
149     store(p);
150     somPersistencePO::PO::
151         uninit_for_object_passivation();
152     somStream::Streamable::
153         uninit_for_object_passivation();
154 }
155

```

Continued

```

156 void MessageQueue::uninit_for_object_move()
157 {
158     somPersistencePO::PO::uninit_for_object_move();
159     somStream::Streamable::uninit_for_object_move();
160 }
161
162 void MessageQueue::externalize_to_stream(
163     CosStream::StreamIO *stream)
164 {
165     stream->write_string(name);
166     stream->write_short(count);
167     for (Mqueue *cur = mq; cur != NULL;
168         cur=cur->next) {
169         stream->write_string(cur->message);
170     }
171     somStream::Streamable::
172         externalize_to_stream(stream);
173 }
174
175 void MessageQueue::internalize_from_stream(
176     CosStream::StreamIO * stream,
177     CosLifeCycle::FactoryFinder *ff)
178 {
179     Clear();
180     if (name)
181         delete name;
182     name = stream->read_string();
183     int number = stream->read_short();
184     for (int i=0; i < number; i++) {
185         char *str = stream->read_string();
186         Send(str);
187     }
188     somStream::Streamable::
189         internalize_from_stream(stream, ff);
190 }

```

**Implementation of POSSOM Class MessageQueueManager
(*possom\mqueue\mqmgr.cpp*):**

```

1  #include <iostream.h>
2  #include <stdio.h>
3  #include <assert.h>
4  #include <somd.hh>
5  #include <somposx.hh>
6  #include "osutil.h"
7  #include "mqmgr.hh"
8
9  #define MAX_QUEUES 20
10
11 static Environment ev =
12     (SOM_InitEnvironment(&__SOMEnv, &ev), ev);
13
14 // workaround for problem in beta
15 static CosLifeCycle::FactoryFinder *savedPF = NULL;

```



```

16
17 MessageQueueManager::MessageQueueManager()
18 {
19     init_for_object_creation();
20     if (! savedFF)
21         savedFF = getFactoryFinder();
22 }
23
24 MessageQueueManager::~MessageQueueManager()
25 {
26     return;
27     // don't delete message queues because will be
28     // passivated and deleted by OS Server
29 }
30
31 MessageQueue *MessageQueueManager::
32     GetMessageQueue(char *name)
33 {
34     for (int i=0; i<MAX_QUEUES; i++)
35         if (mqueues[i] &&
36             mqueues[i]->name &&
37             strcmp(mqueues[i]->name, name) == 0)
38             return mqueues[i];
39     for (i=0; i<MAX_QUEUES && mqueues[i]; i++)
40         ;
41     if (i == MAX_QUEUES)
42         return NULL;
43     if (! (mqueues[i] = new MessageQueue(name))) {
44         return NULL;
45     }
46     store(p); // store with new queue
47     return mqueues[i];
48 }
49
50 MessageQueue *MessageQueueManager::
51     GetMessageQueue(int qnum)
52 {
53     if (qnum < MAX_QUEUES && mqueues[qnum])
54         return mqueues[qnum];
55     else
56         return NULL;
57 }
58
59 SOMObject *MessageQueueManager::init_for_object_creation()
60 {
61     somPersistencePO::PO::init_for_object_creation();
62     somStream::Streamable::init_for_object_creation();
63     for (int i=0; i<MAX_QUEUES; i++)
64         mqueues[i] = NULL;
65     return this;
66 }
67
68 SOMObject *MessageQueueManager::
69     init_for_object_reactivation()
70 {

```

Continued

```

71     somPersistencePO::PO::
72         init_for_object_reactivation();
73     somStream::Streamable::
74         init_for_object_reactivation();
75     for (int i=0; i<MAX_QUEUES; i++)
76         mqueues[i] = NULL;
77     return this;
78 }
79
80 SOMObject *MessageQueueManager::init_for_object_copy()
81 {
82     somPersistencePO::PO::init_for_object_copy();
83     somStream::Streamable::init_for_object_copy();
84     return this;
85 }
86
87 void MessageQueueManager::
88     uninit_for_object_destruction()
89 {
90     for (int i=0; i<MAX_QUEUES; i++) {
91         if (mqueues[i]) {
92             mqueues[i]->Delete(p);
93             mqueues[i]->uninit_for_object_destruction();
94             mqueues[i]->p->somFree();
95             mqueues[i]->somFree();
96         }
97     }
98     somPersistencePO::PO::
99         uninit_for_object_destruction();
100    somStream::Streamable::
101        uninit_for_object_destruction();
102 }
103
104 // Don't call store here because to store this object,
105 // need to be sure that the contained MessageQueues
106 // are still active. They could have already been
107 // passivated by OSServer.
108 void MessageQueueManager::uninit_for_object_passivation()
109 {
110     somPersistencePO::PO::
111         uninit_for_object_passivation();
112     somStream::Streamable::
113         uninit_for_object_passivation();
114 }
115
116 void MessageQueueManager::uninit_for_object_move()
117 {
118     somPersistencePO::PO::uninit_for_object_move();
119     somStream::Streamable::uninit_for_object_move();
120 }
121
122 // Writes the queue count and a stringified reference for
123 // each active queue. Note that this does not store the
124 // queue contents, just a reference to it. Must store
125 // queue separately. Note that the queues must still be in

```

```

126 // memory in order to save reference, ie. they cannot have
127 // been passivated.
128 void MessageQueueManager::externalize_to_stream(
129     CosStream::StreamIO *stream)
130 {
131     int count = 0;
132     for (int i=0; i<MAX_QUEUES; i++)
133         if (mqueues[i])
134             ++count;
135     stream->write_short(count);
136     for (i=0; i<MAX_QUEUES; i++) {
137         if (mqueues[i]) {
138             stream->write_object(mqueues[i]);
139         }
140     }
141     somStream::Streamable::externalize_to_stream(stream);
142 }
143
144 // When each stringified reference is read back
145 // from the stream, this will cause the underlying queue
146 // to be reactivated into memory.
147 void MessageQueueManager::internalize_from_stream(
148     CosStream::StreamIO * stream,
149     CosLifecycle::FactoryFinder *ff)
150 {
151     int count = stream->read_short();
152     if (! ff)
153         // workaround to get around problem in beta
154         ff = savedFF;
155     for (int i=0; i<count; i++) {
156         mqueues[i] = (MessageQueue *)
157             stream->read_object(ff, NULL);
158     }
159     somStream::Streamable::
160         internalize_from_stream(stream, ff);
161 }

```

**Implementation of Class LocalMessageQueueManager
(possom\mqueue\lmqmgr.cpp):**

```

1  #include <iostream.h>
2  #include <stdio.h>
3  #include <limits.h>
4  #include <assert.h>
5
6  #include <sompposx.hh>
7  #include "osutil.h"
8  #include "lmgmgr.h"
9  #include "check.h"
10 #pragma SOMNODataDirect(on)
11 #include "mqmgr.hh"
12 #pragma SOMNODataDirect(off)
13
14 void initServer();

```

Continued

```

15
16 LocalMessageQueueManager::LocalMessageQueueManager()
17 {
18     // initialize DSOM
19     static Environment env;
20     SOM_InitEnvironment(__SOMEnv = &env);
21     SOMD_Init(__SOMEnv);
22     initServer();
23     // Handle memory dealloc with delete/SOMFree
24     SOMD_NoORBfree();
25
26     // resolve to naming context
27     resolveMessageQueueContext();
28     // resolve to MessageQueueManager
29     resolveMessageQueueManager();
30 }
31
32 // release proxies
33 LocalMessageQueueManager::~LocalMessageQueueManager()
34 {
35     {(SOMDObject *)mqmgr->release();
36     {(SOMDObject *)mqueueNC->release();
37 }
38
39 // returns the proxy for the remote MessageQueue
40 // by queue number
41 MessageQueue* LocalMessageQueueManager::
42     GetMessageQueue(int qnum)
43 {
44     return mqmgr->GetMessageQueue(qnum);
45 }
46
47
48 // returns the proxy for the remote MessageQueue by
49 // queue name. Queues are in the name server, so a
50 // name server search is performed first for the
51 // queue before checking the server.
52 MessageQueue* LocalMessageQueueManager::
53     GetMessageQueue(char *qName)
54 {
55     CosNaming::Name name;
56     CosNaming::NameComponent namingComponent;
57     name._length = name._maximum = 1;
58     name._buffer = &namingComponent;
59     name._buffer[0].id = qName;
60     name._buffer[0].kind = NULL;
61
62     // look for the message queue locally first
63     MessageQueue *mq =
64         (MessageQueue *)mqueueNC->resolve(&name);
65     if (checkError(__SOMEnv))
66         mq = NULL;
67     if (!mq) {
68         // file/object doesn't exist, get from server
69         mq = mqmgr->GetMessageQueue(qName);

```

```

70     if (mq) {
71         somPersistencePOSIX::PID_POSIX *pid =
72             (somPersistencePOSIX::PID_POSIX *)
73             somdCreate(__SOMEnv,
74                 "somPersistencePOSIX::PID_POSIX",
75                 TRUE);
76         assert(! checkError(__SOMEnv, TRUE));
77
78         // assign persistent id to the object
79         char *poname = new char[strlen(qName) + 5];
80         assert(poname != NULL);
81         sprintf(poname, "%s.pds", qName);
82         pid->pathName = poname;
83         mq->p = pid;
84
85         // bind to naming service
86         mqueueNC->bind(&name, mq);
87     }
88 }
89 return mq;
90 }
91
92 void LocalMessageQueueManager::
93     resolveMessageQueueContext()
94 {
95     // resolve to root naming context
96     ExtendedNaming::ExtendedNamingContext *rootNC =
97         (ExtendedNaming::ExtendedNamingContext *)
98         SOMD_ORBObject->
99         resolve_initial_references("NameService");
100
101     CosNaming::Name name;
102     CosNaming::NameComponent namingComponent;
103     name._length = name._maximum = 1;
104     name._buffer = &namingComponent;
105     name._buffer[0].id = "MessageQueueContext";
106     name._buffer[0].kind = NULL;
107
108     mqueueNC = (CosNaming::NamingContext *)
109         rootNC->resolve(&name);
110     if (checkError(__SOMEnv))
111         mqueueNC = NULL;
112     else if (mqueueNC &&
113         ! isValidRemoteObject(mqueueNC)) {
114         rootNC->unbind(&name);
115         checkError(__SOMEnv);
116         mqueueNC = NULL;
117     }
118     if (! mqueueNC) {
119         mqueueNC = rootNC->bind_new_context(&name);
120         assert(! checkError(__SOMEnv, TRUE));
121     }
122     ((SOMDObject *)rootNC)->release();
123 }
124

```

Continued

```

125 void LocalMessageQueueManager::
126     resolveMessageQueueManager()
127 {
128     CosNaming::Name name;
129     CosNaming::NameComponent namingComponent;
130     name._length = name._maximum = 1;
131     name._buffer = &namingComponent;
132     name._buffer[0].id = "MessageQueueManager";
133     name._buffer[0].kind = NULL;
134
135     // look for the manager in the naming service
136     mcmgr = (MessageQueueManager *)
137         mqueueNC->resolve(&name);
138     if (checkError(__SOMEnv)) {
139         mcmgr = NULL;
140     } else if (mcmgr && ! isValidRemoteObject(mcmgr)) {
141         mcmgr = NULL;
142         // unbind all in context, since target object bad
143         CosNaming::BindingList list = {0, 0, NULL};
144         CosNaming::BindingIterator *iterator;
145         mqueueNC->list(INT_MAX, &list, &iterator);
146         for (int i = 0; i < list._length; i++) {
147             mqueueNC->
148                 unbind(&list._buffer[i].binding_name);
149         }
150         delete list._buffer;
151     }
152
153     if (! mcmgr) {
154         // Locate factory for MessageQueueManager
155         SOMClass *factory =
156             findFactory("MessageQueueManager");
157
158         // Create a remote object through the factory
159         mcmgr = (MessageQueueManager *)factory->somNew();
160         assert(mcmgr && ! checkError(__SOMEnv, TRUE));
161
162         ((SOMDObject *)factory)->release();
163
164         // assign persistent id to the object
165         somPersistencePOSIX::PID_POSIX *pid =
166             (somPersistencePOSIX::PID_POSIX *)
167             somdCreate(__SOMEnv,
168                 "somPersistencePOSIX::PID_POSIX", TRUE);
169         assert(! checkError(__SOMEnv, TRUE));
170         pid->pathName = "mqsvr.pds";
171         mcmgr->p = pid;
172         mcmgr->store(mcmgr->p);
173         assert(! checkError(__SOMEnv, TRUE));
174
175         mqueueNC->bind(&name, mcmgr);
176         assert(! checkError(__SOMEnv, TRUE));
177     }
178 }
179

```

```

180 // This function is needed only as a workaround for a
181 // problem in the server when reactivating persistent
182 // objects after the server has been restarted.
183 void initServer()
184 {
185     MessageQueueManager *tmp = (MessageQueueManager *)
186     somdCreate(__SOMEnv, "MessageQueueManager", TRUE);
187
188     somPersistencePOSIX::PID_POSIX *pid =
189     (somPersistencePOSIX::PID_POSIX *)
190     somdCreate(__SOMEnv,
191     "somPersistencePOSIX::PID_POSIX", TRUE);
192     assert(! checkError(__SOMEnv, TRUE));
193     pid->pathName = "dummy.po";
194     tmp->p = pid;
195     tmp->store(tmp->p);
196     tmp->Delete(pid);
197     tmp->uninit_for_object_destruction();
198     tmp->somFree();
199     pid->somFree();
200 }

```

Client Program (possom\mqqueue\tstmq.cpp):

```

1  #include <fstream.h>
2  #include <iostream.h>
3  #include <stdio.h>
4  #include <assert.h>
5  #include <somd.hh>
6
7  #include "check.h"
8  #include "mqmgr.h"
9
10 #pragma SOMNoDataDirect(on)
11 #include "mqqueue.hh"
12 #include "mqmgr.hh"
13 #pragma SOMNoDataDirect(off)
14
15 int main(int argc, char *argv[])
16 {
17     LocalMessageQueueManager mqlist;
18
19     for ( ;; ) {
20         char choice, qname[MAX_QUEUE_NAME_LEN],
21         message[MAX_MESSAGE_LEN];
22         char *msgp;
23         MessageQueue *mq;
24         cout << "Enter choice (S)end, (R)ecieve, (N)umber, "
25         "(L)ist, (D)ump, (C)lear, (Q)uit: ";
26         cin >> choice;
27         switch (choice) {
28             case 's': case 'S':
29                 cout << "Enter queue name and message: ";
30                 cin >> qname;

```

Continued

```

31         cin.getline(message, sizeof(message));
32         if ( (mq=mqlist.GetMessageQueue(qname)) != NULL)
33             mq->Send(message);
34         break;
35     case 'r': case 'R':
36         cout << "Enter queue name: ";
37         cin >> qname;
38         if ( (mq=mqlist.GetMessageQueue(qname)) != NULL) {
39             if (mq->Receive(&msgp) == SUCCESS)
40                 cout << "Received message from queue " <<
41                     qname << ": " << msgp << endl;
42             else
43                 cout << "No message from from queue "
44                     << qname << endl;
45         }
46         break;
47     case 'l': case 'L':
48         int i;
49         for (i=0; i < MAX_QUEUES; i++) {
50             if ( (mq=mqlist.GetMessageQueue(i)) != NULL)
51                 cout << "Name: " << mq->name << " count: "
52                     << mq->Count() << endl;
53         }
54         break;
55     case 'd': case 'D':
56         cout << "Enter queue name: ";
57         cin >> qname;
58         if ( (mq=mqlist.GetMessageQueue(qname)) != NULL)
59             mq->Dump();
60         break;
61     case 'n': case 'N':
62         cout << "Enter queue name: ";
63         cin >> qname;
64         if ( (mq=mqlist.GetMessageQueue(qname)) != NULL)
65             cout << "Size of queue " <<
66                 qname << ": " << mq->Count() << endl;
67         break;
68     case 'c': case 'C':
69         cout << "Enter queue name: ";
70         cin >> qname;
71         if ( (mq=mqlist.GetMessageQueue(qname)) != NULL)
72             mq->Clear();
73         break;
74     case 'q': case 'Q':
75         return 0;
76
77     if (msgp)
78         delete msgp;          // delete DSOM-allocated memory
79     if (mq)
80         // deallocate proxy
81         ((SOMDObject *)mq)->release();
82     }
83 }
84 return(0);
85 }

```


Makefile (possom\mqueue\makefile):

```

1  all: mqueue.dll mqmgr.dll tstmq.exe somdimpl.dat
2
3  ICCOPTS = /I ../../include
4
5  mqueue.dll: mqueue.hh mqueue.cpp mqueueui.c
6      icc $(ICCOPTS) /Ti+ /Ge- /B"/NOE" mqueue.cpp \
7          mqueueui.c mqueue.def ../../bin/sommeml.obj
8      implib mqueue.lib mqueue.dll
9
10 mqmgr.dll: mqmgr.hh mqmgr.cpp mqmgri.c
11     icc $(ICCOPTS) /Ti+ /Ge- /B"/NOE" mqmgr.cpp mqmgri.c \
12         mqmgr.def ../../bin/sommeml.obj mqueue.lib
13     implib mqmgr.lib mqmgr.dll
14
15 unbind.exe: unbind.cpp
16     icc $(ICCOPTS) /Ti+ /B"/NOE" unbind.cpp \
17         ../../bin/sommeml.obj
18
19 tstmq.exe: mqueue.hh mqmgr.hh tstmq.cpp lmqmgr.h lmqmgr.cpp
20     icc $(ICCOPTS) /Ti+ /B"/NOE" tstmq.cpp lmqmgr.cpp \
21         ../../bin/sommeml.obj mqueue.lib mqmgr.lib
22
23 mqueue.idl: mqueue.hh
24     icc $(ICCOPTS) mqueue.hh
25     sc -sir -u mqueue.idl
26
27 mqmgr.idl: mqmgr.hh
28     icc $(ICCOPTS) mqmgr.hh
29     sc -sir -u mqmgr.idl
30
31 mqueueui.c: mqueue.idl
32     sc -simod mqueue.idl
33     sc -sdef mqueue.idl
34
35 mqmgri.c: mqmgr.idl
36     sc -simod mqmgr.idl
37     sc -sdef mqmgr.idl
38
39 somdimpl.dat:
40     regimpl -D -i MQServer
41     regimpl -A -i MQServer -p "somosvr.exe " \
42         -v "somOS::Server" -m on
43     regimpl -a -i MQServer -c MessageQueueManager
44     regimpl -a -i MQServer -c MessageQueue
45     regimpl -a -i MQServer -c \
46         somPersistencePOSIX::PID_POSIX
47     regimpl -a -i MQServer -c \
48         somPersistencePOSIX::PDS_POSIX

```


SOMObject Header Files

This appendix contains listings for several of the SOMObject header files that are referenced throughout the book: **<somapi.h>**, **<somcorba.h>**, **<som.hh>**, **<somh.hh>**, **<somobj.hh>**, and **<somcls.hh>**. The versions shipped or generated with the SOMObjects 3.0 beta are shown.

<somapi.h>

```
/*
 * COMPONENT_NAME: somk
 *
 * ORIGINS: 27
 *
 *      25H7912 (C) COPYRIGHT International Business Machines Corp. 1992,
 *      1994, 1996
 *      All Rights Reserved
 *      Licensed Materials - Property of IBM
 *      US Government Users Restricted Rights - Use, duplication or disclosure
 *      restricted by GSA ADP Schedule Contract with IBM Corp.
 */

/*
 * SOMAPI.H
 */
```

```

*   Note: Typedefs, function prototypes, and structure fields declared here
*   explicitly use a star (*) to indicate references to objects that support
*   IDL interfaces. For example, the first argument in the following
*   function prototype is typed as pointer to an object that supports the
*   SOMObject interface:
*
*   somMethodProc * SOMLINK somResolve(SOMObject *obj, somMToken mdata)
*
*   These types reflect the actual implementation of object references
*   in SOM as pointers. The CORBA compliant C language bindings are
*   designed so that programmers don't explicitly indicate pointers when
*   declaring object reference types, but this header is included into
*   the CORBA compliant C language binding files before this capability
*   has been enabled. Thus, at this point in the processing of header files
*   by a C compiler, it is appropriate that the explicit star appear in
*   object reference types. You can view the header file som.h to see how
*   the ability to omit the explicit star is enabled after this header is
*   processed, but, in any case, you can be assured that the typedefs,
*   function prototypes, and structure fields declared here all "work"
*   with the CORBA compliant C language bindings.
*/

#ifndef somapi_h
#define somapi_h

/* SOM Version Numbers */
SOMEXTERN long SOMDLINK SOM_MajorVersion;
SOMEXTERN long SOMDLINK SOM_MinorVersion;

/* SOM Thread Support */
SOMEXTERN long SOMDLINK SOM_MaxThreads;

/*-----
*   Typedefs for pointers to functions
*/

typedef void      SOMLINK somTD_classInitRoutine(SOMClass *, SOMClass *);
typedef int      SOMLINK somTD_SOMOutCharRoutine(char);
typedef int      SOMLINK somTD_SOMLoadModule(IN string /* className */,
                                              IN string /* fileName */,
                                              IN string /* functionName */,
                                              IN long /* majorVersion */,
                                              IN long /* minorVersion */,
                                              OUT somToken /* modHandle */);
typedef int      SOMLINK somTD_SOMDeleteModule(IN somToken /* modHandle */);
typedef string SOMLINK somTD_SOMClassInitFuncName(void);
typedef somToken SOMLINK somTD_SOMMalloc(IN size_t /* nbytes */);
typedef somToken SOMLINK somTD_SOMCalloc(IN size_t /* element_count */,
                                           IN size_t /* element_size */);
typedef somToken SOMLINK somTD_SOMRealloc(IN somToken /* memory */,
                                           IN size_t /* nbytes */);
typedef void      SOMLINK somTD_SOMFree(IN somToken /* memory */);
typedef void      SOMLINK somTD_SOMError(IN int /* code */,

```

```

        IN string /* fileName */;
        IN int /* lineNum */;
typedef unsigned long SOMLINK somTD_SOMCreateMutexSem (OUT somToken *sem);
typedef unsigned long SOMLINK somTD_SOMRequestMutexSem (IN somToken sem);
typedef unsigned long SOMLINK somTD_SOMReleaseMutexSem (IN somToken sem);
typedef unsigned long SOMLINK somTD_SOMDestroyMutexSem (IN somToken sem);
typedef unsigned long SOMLINK somTD_SOMGetThreadId (void);

/*-----
 * Windows extra procedures:
 */

SOMEXTERN void SOMLINK somSetOutChar(somTD_SOMOutCharRoutine *outch);
SOMEXTERN SOMClassMgr * SOMLINK somMainProgram (void);
SOMEXTERN void SOMLINK somEnvironmentEnd (void);
SOMEXTERN boolean SOMLINK somAbnormalEnd (void);

/*-----*/

#ifndef SOM_MDT_DEFINED
/* -- Method/Data Tokens -- For locating methods and data members. */
typedef somToken somMToken;
typedef somToken somDTToken;
#endif

/*-----
 * The Method Table Structure
 */

#ifndef SOM_MTAB_DEFINED
/* -- to specify an embedded object (or array of objects). */
typedef struct {
    SOMClass **copp; /* address of class of object ptr */
    long cnt; /* object count */
    long offset; /* Offset to pointer (to embedded objs) */
} somEmbeddedObjStruct;
#endif

#ifndef SOM_CLASSINFO_DEFINED
typedef somToken somClassInfo;
#endif

typedef struct somMethodTabStruct {
    SOMClass *classObject;
    somClassInfo *classInfo;
    char *className;
    long instanceSize;
    long dataAlignment;
    long mtabSize;
    long protectedDataOffset; /* from class's introduced data */
    somDTToken protectedDataToken;
    somEmbeddedObjStruct *embeddedObjs;
    /* remaining structure is opaque */

```

```

        somMethodProc* entries[1];
    } somMethodTab, *somMethodTabPtr;
#endif

/* -- For building lists of method tables */
typedef
    struct somMethodTabList {
        somMethodTab      *mtab;
        struct somMethodTabList *next;
    } somMethodTabList, *somMethodTabs;

/* -- For building lists of class objects */
typedef
    struct somClassList {
        SOMClass      *cls;
        struct somClassList *next;
    } somClassList, *somClasses;

/* -- For building lists of objects */
typedef struct somObjectList {
    SOMObject *obj;
    struct somObjectList *next;
} somObjectList, *somObjects;
/*-----
 * C++-style constructors are called initializers in SOM. Initializers
 * are methods that receive a pointer to a somCtrlStruct as an argument.
 */

typedef struct {
    SOMClass      *cls; /* the class whose introduced data is to be initialized
    */
    somMethodProc *defaultInit;
    somMethodProc *defaultCopyInit;
    somMethodProc *defaultConstCopyInit;
    somMethodProc *defaultNCArgCopyInit;
    long          dataOffset;
    somMethodProc *legacyInit;
} somInitInfo;

typedef struct {
    SOMClass      *cls; /* the class whose introduced data is to be destroyed */
    somMethodProc *defaultDestruct;
    long          dataOffset;
    somMethodProc *legacyUninit;
} somDestructInfo;

typedef struct {
    SOMClass      *cls; /* the class whose introduced data is to be assigned */
    somMethodProc *defaultAssign;
    somMethodProc *defaultConstAssign;
    somMethodProc *defaultNCArgAssign;
    somMethodProc *udaAssign;

```

```

        somMethodProc *udaConstAssign;
        long          dataOffset;
} somAssignInfo;

typedef octet *somBooleanVector;
#ifdef SOM_CTI_DEFINED
typedef somToken somCtrlInfo;
#endif

typedef struct somInitCtrlStruct {
    somBooleanVector mask; /* an array of booleans to control ancestor calls */
    somInitInfo      *info; /* an array of structs */
    int              infoSize; /* increment for info access */
    somCtrlInfo      ctrlInfo;
} somInitCtrl, som3InitCtrl;

typedef struct somDestructCtrlStruct {
    somBooleanVector mask; /* an array of booleans to control ancestor calls */
    somDestructInfo *info; /* an array of structs */
    int              infoSize; /* increment for info access */
    somCtrlInfo      ctrlInfo;
} somDestructCtrl, som3DestructCtrl;

typedef struct somAssignCtrlStruct {
    somBooleanVector mask; /* an array of booleans to control ancestor calls */
    somAssignInfo    *info; /* an array of structs */
    int              infoSize; /* increment for info access */
    somCtrlInfo      ctrlInfo;
} somAssignCtrl, som3AssignCtrl;

/*-----
 * The Class Data Structures -- these are used to implement static
 * method and data interfaces to SOM objects.
 */

/* -- (Generic) Class data Structure */
typedef struct {
    SOMClass *classObject; /* changed by shadowing */
    somToken tokens[1];    /* method tokens, etc. */
} somClassDataStructure, *somClassDataStructurePtr;

/*
 * A special info access structure pointed to by
 * the parentMtab entry of somCClassDataStructure.
 */
typedef void somTP_somRenewNoInitNoZeroThunk(void*);
#if (defined(__IBMC__) && !defined(__cplusplus))
#pragma linkage(somTP_somRenewNoInitNoZeroThunk, system)
#endif
typedef somTP_somRenewNoInitNoZeroThunk *somTD_somRenewNoInitNoZeroThunk;

typedef struct {
    somMethodTab    *mtab; /* this class' mtab -- changed by shadowing */

```

```

    somMethodTabs    next; /* the parent mtab -- unchanged by shadowing */
    SOMClass         *classObject; /* unchanged by shadowing */
    somTD_somRenewNoInitNoZeroThunk somRenewNoInitNoZeroThunk; /* changed by
    shadowing */
    long             instanceSize; /* changed by shadowing */
    somMethodProc     **initializers; /* resolved initializer array in release-
    order */
    somMethodProc     **resolvedMTokens; /* resolved methods */
    somInitCtrl       initCtrl; /* these fields are filled in if somDTSClass&2 is
    on */
    somDestructCtrl   destructCtrl;
    somAssignCtrl     assignCtrl;
    long              embeddedTotalCount;
    long              hierarchyTotalCount;
    long              unused;
} somParentMtabStruct, *somParentMtabStructPtr;

/*
 * (Generic) Auxiliary Class Data Structure
 */
typedef struct {
    somParentMtabStructPtr parentMtab;
    somDTToken             instanceDataToken;
    somMethodProc           *wrappers[1]; /* for valist methods */
} somCClassDataStructure, *somCClassDataStructurePtr;

/*-----
 * Offset-based method resolution.
 */
SOMEXTERN somMethodProc * SOMLINK somResolve(SOMObject *obj, somMToken mdata);
SOMEXTERN somMethodProc * SOMLINK somParentResolve(somMethodTabs parentMtab,
    somMToken mToken);
SOMEXTERN somMethodProc * SOMLINK somParentNumResolve(somMethodTabs parentMtab,
    int parentNum,
    somMToken mToken);
SOMEXTERN somMethodProc * SOMLINK somClassResolve(SOMClass *, somMToken mdata);
SOMEXTERN somMethodProc * SOMLINK somResolveTerminal(SOMClass *, somMToken
    mdata);
SOMEXTERN somMethodProc * SOMLINK somAncestorResolve(SOMObject *obj, /* the
    object */
    somCClassDataStructure *ccds, /* id
    the ancestor */
    somMToken mToken);
SOMEXTERN somMethodProc * SOMLINK somResolveByName(SOMObject *obj,
    char *methodName);

/*-----
 * Offset-based data resolution
 */
SOMEXTERN somToken SOMLINK somDataResolve(SOMObject *obj, somDTToken dataId);
SOMEXTERN somToken SOMLINK somDataResolveChk(SOMObject *obj, somDTToken dataId);

/*-----
 * Misc. procedures:
 */

```



```

/*
 * Create and initialize the SOM environment
 *
 * Can be called repeatedly
 *
 * Will be called automatically when first object (including a class
 * object) is created, if it has not already been done.
 *
 * Returns the SOMClassMgrObject
 */
SOMEXTERN SOMClassMgr * SOMLINK somEnvironmentNew (void);

/*
 * Test whether <obj> is a valid SOM object. This test is based solely on
 * the fact that (on this architecture) the first word of a SOM object is a
 * pointer to its method table. The test is therefore most correctly understood
 * as returning true if and only if <obj> is a pointer to a pointer to a
 * valid SOM method table. If so, then methods can be invoked on <obj>.
 */
SOMEXTERN boolean SOMLINK somIsObj(somToken obj);

/*
 * Return the class that introduced the method represented by a given method token.
 */
SOMEXTERN SOMClass* SOMLINK somGetClassFromMToken(somMToken mToken);

/*-----
 * String Manager; stem <somasm>
 */
SOMEXTERN somId SOMLINK somCheckId (somId id);
/* makes sure that the id is registered and in normal form, returns */
/* the id */

SOMEXTERN int SOMLINK somRegisterId(somId id);
/* Same as somCheckId except returns 1 (true) if this is the first */
/* time the string associated with this id has been registered, */
/* returns 0 (false) otherwise */

SOMEXTERN somId SOMLINK somIdFromString (string aString);
/* caller is responsible for freeing the returned id with SOMFree */

SOMEXTERN string SOMLINK somStringFromId (somId id);

SOMEXTERN int SOMLINK somCompareIds(somId id1, somId id2);
/* returns true (1) if the two ids are equal, else false (0) */

SOMEXTERN unsigned long SOMLINK somTotalRegIds(void);
/* Returns the total number of ids that have been registered so far, */
/* you can use this to advise the SOM run-time concerning expected */

```

```

/* number of ids in later executions of your program, via a call to */
/* somSetExpectedIds defined below */

SOMEXTERN void SOMLINK somSetExpectedIds(unsigned long numIds);
/* Tells the SOM run-time how many unique ids you expect to use during */
/* the execution of your program, this can improve space and time */
/* utilization slightly, this routine must be called before the SOM */
/* environment is created to have any effect */

SOMEXTERN unsigned long SOMLINK somUniqueKey(somId id);
/* Returns the unique key for this id, this key will be the same as the */
/* key in another id if and only if the other id refers to the same */
/* name as this one */

SOMEXTERN void SOMLINK somBeginPersistentIds(void);
/* Tells the id manager that strings for any new ids that are */
/* registered will never be freed or otherwise modified. This allows */
/* the id manager to just use a pointer to the string in the */
/* unregistered id as the master copy of the ids string. Thus saving */
/* space */
/* Under normal use (where ids are static variables) the string */
/* associated with an id would only be freed if the code module in */
/* which it occurred was unloaded */

SOMEXTERN void SOMLINK somEndPersistentIds(void);
/* Tells the id manager that strings for any new ids that are */
/* registered may be freed or otherwise modified. Therefore the id */
/* manager must copy the strings in order to remember the name of an */
/* id. */

/*-----
 * Class Manager: SOMClassMgr, stem <somcm>
 */

/* Global class manager object */
SOMEXTERN SOMClassMgr * SOMDLINK SOMClassMgrObject;

/* The somRegisterClassLibrary function is provided for use in SOM class
 * libraries on platforms that have loader-invoked entry points associated with
 * shared libraries (DLLs).
 *
 * This function registers a SOM Class Library with the SOM Kernel.
 * The library is identified by its file name and a pointer
 * to its initialization routine. Since this call may occur
 * prior to the invocation of somEnvironmentNew, its actions
 * are deferred until the SOM environment has been initialized.
 * At that time, the SOMClassMgrObject is informed of all
 * pending library initializations via the _somRegisterClassLibrary
 * method. The actual invocation of the library's initialization
 * routine will occur during the execution of the SOM_MainProgram
 * macro (for statically linked libraries), or during the _somFindClass
 * method (for libraries that are dynamically loaded).
 */

```

```

SOMEXTERN void SOMLINK somRegisterClassLibrary (string libraryName,
        somMethodProc *libraryInitRtn);
SOMEXTERN void SOMLINK somUnregisterClassLibrary (string libraryName);

/*-----
 * Method Stubs -- Signature Support
 *
 *
 * This section defines the structures used to pass method signature
 * info to the run time. This supports selection of generic apply stubs
 * and run-time generation of redispachstubs when these are needed. The
 * information is registered with the run time when methods are defined.
 *
 * When calling somAddStaticMethod, if the redispachStub is -1, then a
 * pointer to a struct of type somApRdInfo is passed as the applyStub.
 * Otherwise, the passed redispachstub and applystub are taken as given.
 * When calling somAddDynamicMethod, an actual apply stub must be passed.
 * Redispach stubs for dynamic methods are not available, nor is
 * automated support for dynamic method apply stubs. The following
 * structures only appropriate in relation to static methods.
 *
 * In SOMr2, somAddStaticMethod can be called with an actual redispachstub
 * and applystub *ONLY* if the method doesn't return a structure. Recall
 * that no SOMr1 methods returned structures, so SOMr1 binaries obey this
 * restriction. The reason for this rule is that SOMr2 *may* use thunks,
 * and thunks need to know if a structure is returned. We therefore assume
 * that if no signature information is provided for a method through the
 * somAddStaticMethod interface, then the method returns a scalar.
 *
 * If a structure is returned, then a -1 *must* be passed to
 * somAddStaticMethod as a redispachstub. In any case, if a -1 is passed,
 * then this means that the applystub actually points to a structure of type
 * somApRdInfo. This structure is used to hold and access signature
 * information encoded as follows.
 *
 * If the somApRdInfo pointer is NULL, then, if the run time was built with
 * SOM_METHOD_STUBS defined, a default signature is assumed (no arguments,
 * and no structure returned); otherwise, the stubs are taken as
 * somDefaultMethod (which produces a run-time error when used) if dynamic
 * stubs are not available.
 *
 * If the somApRdInfo pointer is not NULL, then the structure it points to can
 * either include (non-null) redispach and applystubs (the method is then
 * assumed to return a structure), or null stubs followed by information needed
 * to generate necessary stubs dynamically.
 */

typedef unsigned long somRdAppType; /* method signature code -- see def below */
typedef unsigned long somFloatMap[13]; /* float map -- see def below */
typedef struct somMethodInfoStruct {
    somRdAppType    callType;
    long            va_listSize;
    somFloatMap     *float_map;
} somMethodInfo;

```

```

typedef struct somApRdInfoStruct {
    somMethodProc *rdStub;
    somMethodProc *apStub;
    somMethodInfo *stubInfo;
} somApRdInfo;

/*
 * Values for somRdAppType are generated by summing one from column A and one
 * from column B of the following constants:
 */
/* Column A: return type */
#define SOMRdRetSimple 0 /* Return type is a non-float fullword */
#define SOMRdRetFloat 2 /* Return type is (single) float */
#define SOMRdRetDouble 4 /* Return type is double */
#define SOMRdRetLongDouble 6 /* Return type is long double */
#define SOMRdRetAggregate 8 /* Return type is struct or union */
#define SOMRdRetByte 10 /* Return type is a byte */
#define SOMRdRetHalf 12 /* Return type is a (2 byte) halfword */
/* Column B: are there any floating point scalar arguments? */
#define SOMRdNoFloatArgs 0
#define SOMRdFloatArgs 1

/* A somFloatMap is only needed on RS/6000 */
/*
 * This is an array of offsets for up to the first 13 floating point arguments.
 * If there are fewer than 13 floating point arguments, then there will be
 * zero entries following the non-zero entries which represent the float args.
 * A non-zero entry signals either a single- or a double-precision floating point
 * argument. For a double-precision argument, the entry is the stack
 * frame offset. For a single-precision argument the entry is the stack
 * frame offset + 1. For the final floating point argument, add 2 to the
 * code that would otherwise be used.
 */
#define SOMFMSingle 1 /* add to indicate single-precision */
#define SOMFMLast 2 /* add to indicate last floating point arg */

/*-----
 * -- somApply --
 *
 * This routine replaces direct use of applyStubs in SOMr1. The reason
 * for the replacement is that the SOMr1 style of applyStub is not
 * generally available in SOMr2, which uses a fixed set of applyStubs,
 * according to method information in the somMethodData. In particular,
 * neither the redispach stub nor the apply stub found in the method
 * data structure are necessarily useful as such. The method somGetRdStub
 * is the way to get a redispach stub, and the above function is the
 * way to call an apply stub. If an appropriate apply stub for the
 * method indicated by md is available, then this is invoked and TRUE is
 * returned; otherwise FALSE is returned.
 *
 * The va_list passed to somApply *must* include the target object,
 * somSelf, as its first entry, and any single precision floating point
 * arguments being passed to the the method procedure must be
 * represented on the va_list using double precision values. retVal cannot

```

```

* be NULL.
*/

#ifdef SOM_SMD_DEFINED
typedef somToken somSharedMethodData;
#endif

typedef struct somMethodDataStruct {
    somId id;
    long type;          /* 0=static, 1=dynamic 2 onstatic */
    somId descriptor;    /* for use with IR interfaces */
    somMToken mToken;    /* NULL for dynamic methods */
    somMethodPtr method; /* depends on resolution context */
    somSharedMethodData *shared;
} somMethodData, *somMethodDataPtr;

SOMEXTERN boolean SOMLINK somApply(SOMObject *somSelf,
    somToken *retVal,
    somMethodDataPtr mdPtr,
    va_list ap);

/*-----
* -- somBuildClass --
*
* This procedure automates construction of a new class object. A variety of
* special structures are used to allow language bindings to statically define
* the information necessary to specify a class. Pointers to these static
* structures are accumulated into an overall "static class information"
* structure or SCI, passed to somBuildClass. The SCI has evolved over time.
* The current version is defined here.
*/

#define SOM_SCI_LEVEL 4

/* The SCI includes the following information:
*
* The address of a class's ClassData structure is passed.
* This structure should have the external name,
* <className>ClassData. The classObject field should be NULL
* (if it is not NULL, then a new class will not be built). somBuildClass will
* set this field to the address of the new class object when it is built.
*
* The address of the class's auxiliary ClassData structure is passed.
* This structure should have the external name,
* <className>CClassData. The parentMtab field will be set by somBuildClass.
* This field often allows method calls to a class object to be avoided.
*
* The other structures referenced by the static class information (SCI)
* are used to:
*/

/*
* to specify a static method. The methodId used here must be

```

```

* a simple name (i.e., no colons). In all other cases,
* where a somId is used to identify a registered method,
* the somId can include explicit scoping. An explicitly scoped
* method name is called a method descriptor. For example,
* the method introduced by SOMObject as somGetClass has the
* method descriptor "SOMObject::somGetClass". When a
* class is contained in an IDL module, the descriptor syntax
* <moduleName>::<className>::<methodName> can be used. Method
* descriptors can be useful when a class supports different methods
* that have the same name (note: IDL prevents this from occurring
* statically, but SOM itself has no problems with this).
*/

typedef struct somStaticMethodStruct {
    somMToken *classData;
    somId *methodId; /* this must be a simple name (no colons) */
    somId *methodDescriptor;
    somMethodProc *method;
    somMethodProc *redispatchStub;
    somMethodProc *applyStub;
} somStaticMethod_t;

/* to specify an overridden method */
typedef struct somOverrideMethodStruct {
    somId *methodId; /* this can be a method descriptor */
    somMethodProc *method;
} somOverrideMethod_t;

/* to inherit a specific parent's method implementation */
typedef struct somInheritedMethodStruct {
    somId *methodId; /* identify the method */
    long parentNum; /* identify the parent */
    somMToken *mToken; /* for parentNumresolve */
} somInheritedMethod_t;

/* to register a method that has been moved from this */
/* class <cls> upwards in the class hierarchy to class <dest> */
typedef struct somMigratedMethodStruct {
    somMToken *clsMToken; /* points into the <cls> classdata structure */
    /* the method token in <dest> will be copied here */
    somMToken *destMToken; /* points into the <dest> classdata structure */
    /* the method token here will be copied to <cls> */
} somMigratedMethod_t;

/* to specify non-internal data */
typedef struct somNonInternalDataStruct {
    somMToken *classData;
    char *basisForDataOffset;
} somNonInternalData_t;

/* to specify a "procedure" or "classdata" */
typedef struct somProcMethodsStruct {
    somMethodProc **classData, *pEntry;
} somProcMethods_t;

```

```

/* to specify a general method 'action' using somMethodStruct */
/*
the type of action is specified by loading the type field of the
somMethodStruct. There are three bit fields in the overall type:

action (in type & 0xFF)
0: static -- (i.e., virtual) uses somAddStaticMethod
1: dynamic -- uses somAddDynamicMethod (classData==0)
2: nonstatic -- (i.e., nonvirtual) uses somAddMethod
3: udaAssign -- registers a method as the udaAssign (but doesn't add the method)
4: udaConstAssign -- like 3, this doesn't add the method
5: somClassResolve Override (using the class pointed to by *classData)
6: somMToken Override (using the method token pointed to by methodId)
   (note: classData==0 for this)
7: classAllocate -- indicates the default heap allocator for this class.
   If classData == 0, then method is the code address (or NULL)
   If classData != 0, then *classData is the code address.
   No other info required (or used)
8: classDeallocate -- like 7, but indicates the default heap deallocator.
9: classAllocator -- indicates a non default heap allocator for this class.
   like 7, but a methodDescriptor can be given.

=== the following is not currently supported ===
binary data access -- in (type & 0x100), valid for actions 0,1,2,5,6
0: the method procedure doesn't want binary data access
1: the method procedure does want binary data access

aggregate return -- in (type & 0x200), used when binary data access requested
0: method procedure doesn't return a structure
1: method procedure does return a structure
*/

typedef struct somMethodStruct {
    unsigned long type;
    somMToken *classData;
    somId *methodId;
    somId *methodDescriptor;
    somMethodProc *method;
    somMethodProc *redispachStub;
    somMethodProc *applyStub;
} somMethods_t;

/* to specify a varargs function */
typedef struct somVarargsFuncsStruct {
    somMethodProc **classData, *vEntry;
} somVarargsFuncs_t;

/* to specify dynamically computed information (incl. embedded objs) */
typedef struct {
    int    version;           /* 1 for now */
    long   instanceDataSize; /* true size (incl. embedded objs) */
    long   dataAlignment;    /* true alignment */
    somEmbeddedObjStruct *embeddedObjs; /* array end == null copp */
} somDynamicSCT;

```

```

/*
    to specify a DTS class, use the somDTSClass entry in the following
    data structure. This entry is a bit vector interpreted as follows:

    (somDTSClass & 0x0001) == the class is a DTS C++ class
    (somDTSClass & 0x0002) == the class wants the initCtrl entries
                           of the somParentMtabStruct filled in.

*/

/*
 * The Static Class Info Structure passed to somBuildClass
 */
typedef struct somStaticClassInfoStruct {
    unsigned long layoutVersion; /* this struct defines layout version
    SOM_SCILEVEL */
    unsigned long numStaticMethods; /* count of smt entries */
    unsigned long numStaticOverrides; /* count of omt entries */
    unsigned long numNonInternalData; /* count of nit entries */
    unsigned long numProcMethods; /* count of pmt entries */
    unsigned long numVarargsFuncs; /* count of vft entries */
    unsigned long majorVersion;
    unsigned long minorVersion;
    unsigned long instanceDataSize; /* instance data introduced by this class
    */
    unsigned long maxMethods; /* count numStaticMethods and numMethods
    */
    unsigned long numParents;
    somId classId;
    somId explicitMetaId;
    long implicitParentMeta;
    somId *parents;
    somClassDataStructure *cds;
    somCClassDataStructure *ccds;
    somStaticMethod_t *smt; /* basic "static" methods for mtab */
    somOverrideMethod_t *omt; /* overrides for mtab */
    char *nitReferenceBase;
    somNonInternalData_t *nit; /* datatokens for instance data */
    somProcMethods_t *pmt; /* Arbitrary ClassData members */
    somVarargsFuncs_t *vft; /* varargs stubs */
    somTP_somClassInitFunc *cif; /* class init function */
    /* end of layout version 1 */

    /* begin layout version 2 extensions */
    long dataAlignment; /* the desired byte alignment for instance data */
    /* end of layout version 2 */

#define SOMSCIVERSION 1

    /* begin layout version 3 extensions */
    long numDirectInitClasses;
    somId *directInitClasses;
    unsigned long numMethods; /* general (including nonstatic) methods for mtab
    */

```



```

    somMethods_t          *mt;
    unsigned long protectedDataOffset; /* access = resolve(instanceDataToken) +
    offset */
    unsigned long somSCIVersion; /* used during development. currently = 1 */
    unsigned long numInheritedMethods;
    somInheritedMethod_t *imt; /* inherited method implementations */
    unsigned long numClassDataEntries; /* should always be filled in */
    somId *classDataEntryNames; /* either NULL or ptr to an array of somIds */
    unsigned long numMigratedMethods;
    somMigratedMethod_t *mmt; /* migrated method implementations */
    unsigned long numInitializers; /* the initializers for this class */
    somId *initializers; /* in order of release */
    unsigned long somDTSClass; /* used to identify a DirectToSOM class */
    somDynamicSCI *dsci; /* used to register dynamically computed info */
    /* end of layout version 3 */

} somStaticClassInfo, *somStaticClassInfoPtr;

SOMEXTERN SOMClass * SOMLINK somBuildClass (
    long inherit_vars,
    somStaticClassInfo *sci,
    long majorVersion,
    long minorVersion);

/*
The arguments to somBuildClass are as follows:

inherit_vars: a bit mask used to control inheritance of implementation
Implementation is inherited from parent i iff the bit 1<i is on, or i>=32.

sci: the somStaticClassInfo defined above.
majorVersion, minorVersion: the version of the class implementation.

*/

/*-----
* Used by old single-inheritance emitters to make class creation
* an atomic operation. Kept for backwards compatibility.
*/
SOMEXTERN void SOMLINK somConstructClass (
    somId classInitRoutine *classInitRoutine,
    SOMClass *parentClass,
    SOMClass *metaClass,
    somClassDataStructure *cds);

/*
* Uses <SOMOutCharRoutine> to output its arguments under control of the ANSI C
* style format. Returns the number of characters output.
*/
SOMEXTERN int SOMLINK somPrintf (string fmt, ...);
/*
* vprintf form of somPrintf
*/

```

```

SOMEXTERN int SOMLINK somVprintf (string fmt, va_list ap);
/*
 * Outputs (via somPrintf) blanks to prefix a line at the indicated level
 */
SOMEXTERN void SOMLINK somPrefixLevel (long level);
/*
 * Combines somPrefixLevel and somPrintf
 */
SOMEXTERN int SOMLINK somLPrintf (int level, string fmt, ...);

/*
 * Replaceable character output handler.
 * Points to the character output routine to be used in development
 * support. Initialized to <somOutChar>, but may be reset at anytime.
 * Should return 0 (false) if an error occurs and 1 (true) otherwise.
 */

SOMEXTERN somTD_SOMOutCharRoutine * SOMDLINK SOMOutCharRoutine;

/*-----
 * Pointers to routines used to do dynamic code loading and deleting
 */

SOMEXTERN somTD_SOMLoadModule * SOMDLINK SOMLoadModule;
SOMEXTERN somTD_SOMDeleteModule * SOMDLINK SOMDeleteModule;
SOMEXTERN somTD_SOMClassInitFuncName * SOMDLINK SOMClassInitFuncName;

/*-----
 * Replaceable SOM Memory Management Interface
 *
 * External procedure variables SOMCalloc, SOMFree, SOMMalloc, SOMRealloc
 * have the same interface as their standard C-library analogs.
 */

SOMEXTERN somTD_SOMCalloc * SOMDLINK SOMCalloc;
SOMEXTERN somTD_SOMFree * SOMDLINK SOMFree;
SOMEXTERN somTD_SOMMalloc * SOMDLINK SOMMalloc;
SOMEXTERN somTD_SOMRealloc * SOMDLINK SOMRealloc;

/*-----
 * Replaceable SOM Error handler
 */

SOMEXTERN somTD_SOMError * SOMDLINK SOMError;

/*-----
 * Replaceable SOM Semaphore Operations
 *
 * These operations are used by the SOM Kernel to make thread-safe
 * state changes to internal resources.
 */

SOMEXTERN somTD_SOMCreateMutexSem * SOMDLINK SOMCreateMutexSem;
SOMEXTERN somTD_SOMRequestMutexSem * SOMDLINK SOMRequestMutexSem;

```

```

SOMEXTERN somTD_SOMReleaseMutexSem * SOMDLINK SOMReleaseMutexSem;
SOMEXTERN somTD_SOMDestroyMutexSem * SOMDLINK SOMDestroyMutexSem;

/*-----
 * Replaceable SOM Thread Identifier Operation
 *
 * This operation is used by the SOM Kernel to index data unique to the
 * currently executing thread. It must return a small integer that
 * uniquely represents the current thread within the current process.
 */
SOMEXTERN somTD_SOMGetThreadId * SOMDLINK SOMGetThreadId;

/*-----
 * Externals used in the implementation of SOM, but not part of the
 * SOM API.
 */

SOMEXTERN SOMObject * SOMLINK somTestCls(SOMObject *obj, SOMClass*classObj,
                                           string fileName, int lineNumber);
SOMEXTERN void SOMLINK somTest(int condition, int severity, string fileName,
                                int lineNum, string msg);
SOMEXTERN void SOMLINK somAssert(int condition, int ecode,
                                  string fileName, int lineNum, string msg);

#endif /* somapi_h */

```

<somcorba.h>

```

/*
 * COMPONENT_NAME: somk
 *
 * ORIGINS: 27
 *
 * 25H7912 (C) COPYRIGHT International Business Machines Corp.1992,1994,
 * 1996
 * All Rights Reserved
 * Licensed Materials - Property of IBM
 * US Government Users Restricted Rights - Use, duplication or
 * disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
 */
/* @(#) somk/somcorba.h 2.10.1.3 12/26/95 16:48:02 [2/19/96 20:59:13] */

/*
 */

/*
 * SOMCORBA.H
 * Typedefs, structs, & prototypes in support of CORBA extensions
 * to the SOM run-time
 */

```

```

#ifndef SOMCorba_h
#define SOMCorba_h
#include <string.h> /* needed for memset, used below */

#ifndef CORBA_FUNCTION_NAMES
#define CORBAObject Object
#endif

/* in SOM, a CORBA object is a SOM object */
typedef SOMObject CORBAObject;

/* CORBA 5.7, p.89 */
#ifndef SOM_BOOLEAN
#define SOM_BOOLEAN
typedef unsigned char boolean;
#endif /* SOM_BOOLEAN */
typedef unsigned char octet;
typedef char *string;

/* CORBA 7.5.1, p. 129 */
typedef string Identifier;

/* CORBA 4.13, p. 80 */
/*
 * Generated SOM usage bindings for IDL enums start at 1, but
 * somcorba.h is not generated, and the original SOM 2.0 somcorba.h
 * used C enum typedefs to define the exception_type and
 * completion_status enumerations. As a result, to maintain backwards
 * binary compatibility, the mapping for these enums starts at 0
 * (which is also the mapping specified by CORBA 2.0).
 *
 * The additional value enum_name_MAX is needed to ensure that all
 * compilers will allocate 4 bytes for these enums. This technique
 * for representing IDL enums is used in the CORBA 2.0 * C++ mappings.
 */
typedef enum exception_type {NO_EXCEPTION, USER_EXCEPTION, SYSTEM_EXCEPTION,
    exception_type_MAX = 2147483647 /* ensure mapped as 4 bytes */
} exception_type;
typedef enum completion_status {YES, NO, MAYBE,
    completion_status_MAX = 2147483647 /* ensure mapped as 4 bytes */
} completion_status;

typedef struct StExcep {
    unsigned long minor;
    completion_status completed;
} StExcep;

typedef struct Environment {
    exception_type _major;
    struct {
        char * _exception_name;
        void * _params;
    } exception;
}

```

```

    void *                _somedAnchor;
) Environment;

/* CORBA 7.6.1, p.139 plus 5.7, p.89 enum Data Type Mapping */
typedef unsigned long TCKind;
#define TypeCode_tk_null      1
#define TypeCode_tk_void      2
#define TypeCode_tk_short     3
#define TypeCode_tk_long      4
#define TypeCode_tk_ushort     5
#define TypeCode_tk_ulong      6
#define TypeCode_tk_float      7
#define TypeCode_tk_double     8
#define TypeCode_tk_boolean    9
#define TypeCode_tk_char      10
#define TypeCode_tk_octet     11
#define TypeCode_tk_any        12
#define TypeCode_tk_TypeCode  13
#define TypeCode_tk_Principal  14
#define TypeCode_tk_objref     15
#define TypeCode_tk_struct     16
#define TypeCode_tk_union      17
#define TypeCode_tk_enum       18
#define TypeCode_tk_string     19
#define TypeCode_tk_sequence   20
#define TypeCode_tk_array      21
#define TypeCode_tk_pointer    101 /* SOM extension */
#define TypeCode_tk_self       102 /* SOM extension */
#define TypeCode_tk_foreign    103 /* SOM extension */

/* Short forms of tk <x> enumerators */

#define tk_null      TypeCode_tk_null
#define tk_void      TypeCode_tk_void
#define tk_short     TypeCode_tk_short
#define tk_long      TypeCode_tk_long
#define tk_ushort    TypeCode_tk_ushort
#define tk_ulong     TypeCode_tk_ulong
#define tk_float     TypeCode_tk_float
#define tk_double    TypeCode_tk_double
#define tk_boolean   TypeCode_tk_boolean
#define tk_char      TypeCode_tk_char
#define tk_octet     TypeCode_tk_octet
#define tk_any       TypeCode_tk_any
#define tk_TypeCode  TypeCode_tk_TypeCode
#define tk_Principal TypeCode_tk_Principal
#define tk_objref    TypeCode_tk_objref
#define tk_struct    TypeCode_tk_struct
#define tk_union     TypeCode_tk_union
#define tk_enum      TypeCode_tk_enum
#define tk_string    TypeCode_tk_string
#define tk_sequence  TypeCode_tk_sequence
#define tk_array     TypeCode_tk_array

```

```

#define tk_pointer      TypeCode_tk_pointer
#define tk_self         TypeCode_tk_self
#define tk_foreign      TypeCode_tk_foreign

#ifndef SOM_TYPECODE
#define SOM_TYPECODE
typedef void * TypeCode;
#endif /* SOM_TYPECODE */

/* CORBA 5.7, p.89 */
typedef struct any {
    TypeCode _type;
    void * _value;
} any;

/* Convenience macros for sequences */
#define sequence(type) _IDL_SEQUENCE_ ## type
#define SOM_SEQUENCE_NAME(name,type)\
    struct name {\
        unsigned long _maximum;\
        unsigned long _length;\
        type * _buffer;\
    }
#define SOM_SEQUENCE(type)\
    struct {\
        unsigned long _maximum;\
        unsigned long _length;\
        type * _buffer;\
    }

#define SOM_SEQUENCE_TPEDEF(type) typedef SOM_SEQUENCE(type) sequence(type)
#define SOM_SEQUENCE_TPEDEF_NAME(type, name) typedef SOM_SEQUENCE(type) name

/* per CORBA 5.10, p.91 */
#ifndef _IDL_SEQUENCE_void_defined
#define _IDL_SEQUENCE_void_defined
SOM_SEQUENCE_TPEDEF (void);
#endif /* _IDL_SEQUENCE_void_defined */

/* SOM extensions for sequence manipulation */
#define GENERIC_SEQUENCE      sequence(void)
#define sequenceNew(type,max) (*(sequence(type) *)tcSequenceNew(TC_##type,max))
/* Note that sequenceNew macro assumes that a TypeCode constant of the */
/* form TC_xxx exists for any type xxx used as an argument */
#define sequenceElement(s,elem) ((s)._buffer[elem])
#define sequenceLength(s)      ((s)._length)
#define sequenceMaximum(s)     ((s)._maximum)

#ifdef __IBMC__
#pragma linkage (somExceptionId, system)
#pragma linkage (somExceptionValue, system)
#pragma linkage (somExceptionFree, system)

```

```

#pragma linkage (somSetException,      system)
#pragma linkage (somGetGlobalEnvironment, system)
#endif /* __IBMC__ */

SOMEXTERN char * SOMLINK somExceptionId (Environment *ev);
SOMEXTERN void * SOMLINK somExceptionValue (Environment *ev);
SOMEXTERN void SOMLINK somExceptionFree (Environment *ev);
SOMEXTERN void SOMLINK somSetException (Environment *ev,
    exception_type major, char *exception_name, void *params);
SOMEXTERN Environment * SOMLINK somGetGlobalEnvironment (void);

/* Exception function names per CORBA 5.19, p.99 */
#define exception_id      somExceptionId
#define exception_value  somExceptionValue
#define exception_free    somExceptionFree

#ifdef TRUE
#define TRUE 1
#endif /* TRUE */
#ifdef FALSE
#define FALSE 0
#endif /* FALSE */

#define SOM_InterfaceRepository\
    (__get_somInterfaceRepository(SOMClassMgrObject))

/* Convenience macros for manipulating environment structures
 *
 * SOM_CreateLocalEnvironment returns a pointer to an Environment.
 * The other 3 macros all expect a single argument that is also
 * a pointer to an Environment. Use the create/destroy forms for
 * a dynamic local environment and the init/uninit forms for a stack-based
 * local environment.
 *
 * For example
 *
 * Environment *ev;
 * ev = SOM_CreateLocalEnvironment ();
 * ... Use &ev in methods
 * SOM_DestroyLocalEnvironment (ev);
 *
 * or
 *
 * Environment ev;
 * SOM_InitEnvironment (&ev);
 * ... Use &ev in methods
 * SOM_UninitEnvironment (&ev);
 */
#define SOM_CreateLocalEnvironment ()\
    ((Environment *) SOMCalloc (1, sizeof (Environment)))
#define SOM_DestroyLocalEnvironment(ev)\
    (somExceptionFree ((ev)), (somGetGlobalEnvironment() == (ev)) ?\

```

```

        (void) 0 : SOMFree ((ev))
#define SOM_InitEnvironment(ev)\
    ((somGetGlobalEnvironment() == (ev)) ?\
        (void *) NULL : memset (((char *) {(ev)}), 0, sizeof {Environment}))
#define SOM_UninitEnvironment(ev)\
    (somExceptionFree ((ev)))

#endif /* SOMCorba_h */

```

<som.hh>

```

/****Start!****
*
*   ORIGINS: 27
*
*   25H7912 (C) COPYRIGHT International Business Machines Corp.
*1992,1994,1996,1996
*   All Rights Reserved
*   Licensed Materials - Property of IBM
*   The source code for this program is not published or otherwise divested
*   of its trade secrets, irrespective of what has been deposited with the
*   U.S. Copyright Office.
*
****End!****/

// som.hh for DTS C++
// SHD: May 27/94

#ifdef SOM_HH_DTS_Included
#define SOM_HH_DTS_Included

#include <somh.hh>
#include <somobj.hh>
#include <somcls.hh>
#include <somcm.hh>

#endif /* SOM_HH_DTS_Included */

```

<somh.hh>

```

/****Start!****
*
*   ORIGINS: 27
*
*   25H7912 (C) COPYRIGHT International Business Machines Corp.
*1992,1994,1996,1996
*   All Rights Reserved
*   Licensed Materials - Property of IBM
*   The source code for this program is not published or otherwise divested

```



```

*   of its trade secrets, irrespective of what has been deposited with the
*   U.S. Copyright Office.
*
***!End!***

// som.hh for DTS C++ (Vx.y)
// SE: May 19/94

#ifndef SOMH_HH_DTS_Included
#define SOMH_HH_DTS_Included

#pragma SOM

#pragma SOMAsDefault(On)
class SOMObject;
class SOMClass;
class SOMClassMgr;
#pragma SOMAsDefault(Pop)

#pragma SOMAsDefault(off)
#include <somltype.h> // linkage definitions
#include <sombtype.h> // base SOM types
#include <somcorba.t> // CORBA types
#include <somapi.h> // API types and function interfaces
#pragma SOMAsDefault(pop)

#endif /* SOMH_HH_DTS_Included */

```

<somobj.hh>

```

#ifndef _DTS_HH_INCLUDED_somobj
#define _DTS_HH_INCLUDED_somobj

/* Start Interface SOMObject */

// This file was generated by the IBM "DirectToSOM" emitter for C++ (V1.121)
// Generated at 04/15/96 05:28:41 EDT
// The efw file is version 1.62

#include <somh.hh>

#pragma SOMAsDefault(on)
class SOMClass;
#pragma SOMAsDefault(pop)
#pragma SOMAsDefault(on)
class SOMObject;
#pragma SOMAsDefault(pop)
#ifndef _IDL_SEQUENCE_SOMObject_defined
#define _IDL_SEQUENCE_SOMObject_defined
#pragma SOMAsDefault(on)

```

```

class SOMObject;
#pragma SOMAsDefault(pop)
typedef struct {
    unsigned long _maximum;
    unsigned long _length;
    SOMObject **_buffer;
} _IDL_SEQUENCE_SOMObject;
#endif // _IDL_SEQUENCE_SOMObject_defined
#ifndef _IDL_SEQUENCE_octet_defined
#define _IDL_SEQUENCE_octet_defined
typedef struct {
    unsigned long _maximum;
    unsigned long _length;
    octet *_buffer;
} _IDL_SEQUENCE_octet;
#endif // _IDL_SEQUENCE_octet_defined
#pragma SOMNonDTS(on)
#pragma SOMAsDefault(on)
class SOMObject {

// This is the SOM root class, all SOM classes must be descended from
// <SOMObject>. <SOMObject> has no instance data so there is no
// per-instance cost to to being descended from it.
#ifdef __SOM_MODULES__
    #pragma SOMClassName(*, "SOMObject")
#else
    #pragma SOMClassName(*, "SOMObject")
#endif
    #pragma SOMNoMangling(*)
    #pragma SOMNonDTS(*)

    #pragma SOMClassVersion (SOMObject, 1, 5)
    #pragma SOMCallstyle (oidl)
    #pragma SOMAsDefault(off)

public:
    typedef _IDL_SEQUENCE_SOMObject SOMObjectSequence;
    typedef _IDL_SEQUENCE_octet BooleanSequence;

    // a structure to describe an object-related offset
    typedef struct somObjectOffset {
        ::SOMObject* obj;
        long offset;
    } somObjectOffset;
#ifndef SOMObject__IDL_SEQUENCE_somObjectOffset_defined
#define SOMObject__IDL_SEQUENCE_somObjectOffset_defined
    struct somObjectOffset;
    typedef struct {
        unsigned long _maximum;
        unsigned long _length;
        somObjectOffset * buffer;
    } _IDL_SEQUENCE_somObjectOffset;
#endif // SOMObject__IDL_SEQUENCE_somObjectOffset_defined
    typedef _IDL_SEQUENCE_somObjectOffset somObjectOffsets;

```

```

#pragma SOMAsDefault(pop)
SOMObject();
SOMObject(SOMObject&);
SOMObject(const SOMObject&);
SOMObject(volatile SOMObject&);
SOMObject(const volatile SOMObject&);
virtual ~SOMObject();

#ifdef __EXTENDED_SOM_ASSIGNMENTS__
virtual SOMObject& somAssign(SOMObject&);
virtual SOMObject& somAssign(const SOMObject&);
virtual SOMObject& somAssign(volatile SOMObject&);
virtual SOMObject& somAssign(const volatile SOMObject&);
#else
virtual SOMObject* somDefaultAssign(som3AssignCtrl*, SOMObject*);
virtual SOMObject* somDefaultConstAssign(som3AssignCtrl*, const SOMObject*);
virtual SOMObject* somDefaultVAssign(som3AssignCtrl*, volatile SOMObject*);
virtual SOMObject* somDefaultConstVAssign(som3AssignCtrl*, const volatile
SOMObject*);
#endif

// Obsolete but still supported. Override somDefaultInit instead of somInit.
virtual void somInit();

// Obsolete but still supported. Override somDestruct instead of somUninit.
virtual void somUninit();

// Use as directed by framework implementations.
virtual void somFree();

// Return the name of the receiver's class.
virtual string somGetClassName();

// Return the receiver's class.
virtual ::SOMClass* somGetClass();

// Returns 1 (true) if the receiver responds to methods
// introduced by <aClassObj>, and 0 (false) otherwise.
virtual boolean somIsA(::SOMClass* aClassObj);

// Returns 1 (true) if the indicated method can be invoked
// on the receiver and 0 (false) otherwise.
virtual boolean somRespondsTo(::somId mId);

// Returns 1 (true) if the receiver is an instance of
// <aClassObj> and 0 (false) otherwise.
virtual boolean somIsInstanceOf(::SOMClass* aClassObj);

// Return the size of the receiver.
virtual long somGetSize();

// Uses <SOMOutCharRoutine> to write a detailed description of this object
// and its current state.
//

```

```

// <level> indicates the nesting level for describing compound objects
// it must be greater than or equal to zero. All lines in the
// description will be preceded by <2*level> spaces.
//
// This routine only actually writes the data that concerns the object
// as a whole, such as class, and uses <somDumpSelfInt> to describe
// the object's current state. This approach allows readable
// descriptions of compound objects to be constructed.
//
// Generally it is not necessary to override this method, if it is
// overridden it generally must be completely replaced.
virtual void somDumpSelf(long level);

// Uses <SOMOutCharRoutine> to write in the current state of this object.
// Generally this method will need to be overridden. When overriding
// it, begin by calling the parent class form of this method and then
// write in a description of your class's instance data. This will
// result in a description of all the object's instance data going
// from its root ancestor class to its specific class.
virtual void somDumpSelfInt(long level);

// Uses <SOMOutCharRoutine> to write a brief string with identifying
// information about this object. The default implementation just gives
// the object's class name and its address in memory.
// <self> is returned.
virtual ::SOMObject* somPrintSelf();

// Obsolete. Use somDispatch instead.
virtual void somDispatchV(::somId methodId, ::somId descriptor,
                        ::va_list ap);

// Obsolete. Use somDispatch instead.
virtual long somDispatchL(::somId methodId, ::somId descriptor,
                        ::va_list ap);

// Obsolete. Use somDispatch instead.
virtual void* somDispatchA(::somId methodId, ::somId descriptor,
                        ::va_list ap);

// Obsolete. Use somDispatch instead.
virtual double somDispatchD(::somId methodId, ::somId descriptor,
                        ::va_list ap);

// This method provides a generic, class-specific dispatch mechanism.
// It accepts as input <retValue> a pointer to the memory area to be
// loaded with the result of dispatching the method indicated by
// <methodId> using the arguments in <ap>. <ap> contains the object
// on which the method is to be invoked as the first argument.
virtual boolean somDispatch(::somToken* retValue, ::somId methodId,
                        ::va_list ap);

// Like somDispatch, but method resolution for static methods is done
// according to the clsObj instance method table.

```

```

virtual boolean somClassDispatch(::SOMClass* clsObj, ::somToken* retValue,
                                ::somId methodId, ::va_list ap);

// cast the receiving object to cls (which must be an ancestor of the
// objects true class. Returns true on success.
virtual boolean somCastObj(::SOMClass* cls);

// reset an object to its true class. Returns true always.
virtual boolean somResetObj();
#pragma SOMReleaseOrder ( \
    "somInit", \
    "somUninit", \
    "somFree", \
    SOMObject(volatile SOMObject&), \
    "somGetClassName", \
    "somGetClass", \
    "somIsA", \
    "somRespondsTo", \
    "somIsInstanceOf", \
    "somGetSize", \
    "somDumpSelf", \
    "somDumpSelfInt", \
    "somPrintSelf", \
    SOMObject(const volatile SOMObject&), \
    "somDispatchV", \
    "somDispatchL", \
    "somDispatchA", \
    "somDispatchD", \
    "somDispatch", \
    "somClassDispatch", \
    "somCastObj", \
    "somResetObj", \
    SOMObject(), \
    SOMObject::~SOMObject(), \
    *, \
    *, \
    SOMObject(SOMObject&), \
    SOMObject(const SOMObject&), \
    "somDefaultAssign", \
    "somDefaultConstAssign", \
    "somDefaultVAssign", \
    "somDefaultConstVAssign", \
    *, \
    *, \
    *, \
    *, \
    *, \
    *, \
    *, \
    *)
};

#pragma SOMAsDefault(pop)
#pragma SOMNonDTS(pop)

```

```
/* End SOMObject */
#endif /* _DTS_HH_INCLUDED_somobj */
```

<somcls.hh>

```
#ifndef _DTS_HH_INCLUDED_somcls
#define _DTS_HH_INCLUDED_somcls

/* Start Interface SOMClass */

// This file was generated by the IBM "DirectToSOM" emitter for C++ (V1.121)
// Generated at 04/15/96 05:26:11 EDT
// The efw file is version 1.62

#include <som.hh>

#pragma SOMAsDefault(on)
class SOMClass;
#pragma SOMAsDefault(pop)
#pragma SOMAsDefault(on)
class SOMObject;
#pragma SOMAsDefault(pop)
#ifdef _IDL_SEQUENCE_SOMClass_defined
#define _IDL_SEQUENCE_SOMClass_defined
#pragma SOMAsDefault(on)
class SOMClass;
#pragma SOMAsDefault(pop)
typedef struct {
    unsigned long _maximum;
    unsigned long _length;
    SOMClass **_buffer;
} _IDL_SEQUENCE_SOMClass;
#endif // _IDL_SEQUENCE_SOMClass_defined
#include <somobj.hh>
#pragma SOMNonDTS(on)

class SOMClass : public ::SOMObject {
//
// This is the SOM metaclass. That is, the instances of this class
// are class objects. When the SOM environment is created an instance
// of SOMClass is created and a pointer to it is placed in the external
// data location (SOMClassClassData.classObject). Bindings provide the
// macro _SOMClass for this expression. _SOMClass is unique in that it
// is its own class object. I.e., _SOMClass == _somGetClass(_SOMClass).
// SOMClass can be subclassed just like any SOM class. The subclasses
// of SOMClass are new metaclasses and can generate class objects with
// different implementations than those produced by _SOMClass.
//
// An important rule for metaclass programming is that no methods
// introduced by SOMClass should ever be overridden. While this
// limits the utility of metaclass programming in SOM, it guarantees
// that SOM will operate correctly. Special class frameworks may be
```

```

// available from IBM to alleviate this restriction. Also, the
// restriction may be lifted in the future.
//

#if defined(_SOM_MODULES__)
#pragma SOMClassName(*, "SOMClass")
#else
#pragma SOMClassName(*, "SOMClass")
#endif

#pragma SOMNoMangling(*)
#pragma SOMNonDTS(*)

#pragma SOMClassVersion (SOMClass, 1, 5)
#pragma SOMCallstyle (oldl)
#pragma SOMAsDefault(off)

public :
#ifndef SOMClass__IDL_SEQUENCE_somToken_defined
#define SOMClass__IDL_SEQUENCE_somToken_defined
typedef struct {
    unsigned long _maximum;
    unsigned long _length;
    somToken *_buffer;
} _IDL_SEQUENCE_somToken;
#endif // SOMClass__IDL_SEQUENCE_somToken_defined
typedef _IDL_SEQUENCE_somToken somTokenSequence;
typedef _IDL_SEQUENCE_SOMClass SOMClassSequence;

// a structure to describe a class-related offset
typedef struct somOffsetInfo {
    ::SOMClass* cls;
    long offset;
} somOffsetInfo;
#ifndef SOMClass__IDL_SEQUENCE_somOffsetInfo_defined
#define SOMClass__IDL_SEQUENCE_somOffsetInfo_defined
struct somOffsetInfo;
typedef struct {
    unsigned long _maximum;
    unsigned long _length;
    somOffsetInfo *_buffer;
} _IDL_SEQUENCE_somOffsetInfo;
#endif // SOMClass__IDL_SEQUENCE_somOffsetInfo_defined
typedef _IDL_SEQUENCE_somOffsetInfo somOffsets;
#ifndef SOMClass__IDL_SEQUENCE_somId_defined
#define SOMClass__IDL_SEQUENCE_somId_defined
typedef struct {
    unsigned long _maximum;
    unsigned long _length;
    somId *_buffer;
} _IDL_SEQUENCE_somId;
#endif // SOMClass__IDL_SEQUENCE_somId_defined
typedef _IDL_SEQUENCE_somId somIdSequence;
somOffsets somInstanceDataOffsets;

```

```

#pragma SOMAttribute(sominstanceDataOffsets, readonly, virtualaccessors,
nodata)
#pragma SOMAsDefault(pop)
SOMClass();
SOMClass(SOMClass&);
virtual ~SOMClass();
#ifdef __EXTENDED_SOM_ASSIGNMENTS__
virtual SOMClass& somAssign(SOMClass&);
#else
virtual SOMObject* somDefaultAssign(som3AssignCtrl*, SOMObject*);
#endif

// Uses SOMMalloc to allocate storage for a new instance of the
// receiving class, and then calls somRenewNoInitNoZero to load the
// new object's method table pointer. Then somDefaultInit is called to
// initialize the new object. Note: If the instance is a class object,
// somInitMIClass must then be invoked to declare parents and
// initialize the class's instance method table. Upon failure, NULL
// is returned.
virtual ::SOMObject* somNew();

// Equivalent to somNew except that storage is not allocated.
// <obj> is taken as the address of the new object.
virtual ::SOMObject* somRenew(void* obj);

// somInitClass is obsolete, and should no longer be used. The SOM 2.0
// kernel provides special handling for redispatch stubs in the case
// of SOM 1.0 classes, and use of this method is what tells the kernel
// that old-style redispatch stubs will be registered.
virtual void somInitClass(string className, ::SOMClass* parentClass,
                          long dataSize, long maxStaticMethods,
                          long majorVersion, long minorVersion);

// This method is invoked when all of the static initialization for
// the class has been finished. The default implementation simply
// registers the newly constructed class with the SOMClassMgr.
virtual void somClassReady();

// This object's class name as a NULL terminated string.
virtual string somGetName();

// Returns the parent class of self (along its "left-hand" derivation
// path), if one exists and NULL otherwise.
virtual ::SOMClass* somGetParent();

// Returns 1 (true) if <self> is a descendent class of <aClassObj> and
// 0 (false) otherwise. Note: a class object is considered to be
// descended itself for the purposes of this method.
virtual boolean somDescendedFrom(::SOMClass* aClassObj);

// Returns 1 (true) if the implementation of this class is
// compatible with the specified major and minor version number and
// false (0) otherwise. An implementation is compatible with the

```



```

// specified version numbers if it has the same major version number
// and a minor version number that is equal to or greater than
// <minorVersion>. The major, minor version number pair (0,0) is
// considered to match any version. This method is usually called
// immediately after creating the class object to verify that a
// dynamically loaded class definition is compatible with a using
// application.
virtual boolean somCheckVersion(long majorVersion, long minorVersion);

// Finds the method procedure associated with <methodId> for this
// class and sets <m> to it. 1 (true) is returned when the
// method procedure is a static method and 0 (false) is returned
// when the method procedure is dynamic method.
//
// If the class does not support the specified method then
// <m> is set to NULL and the return value is meaningless.
//
virtual boolean somFindMethod(::somId methodId, ::somMethodPtr* m);
// Just like <somFindMethod> except that if the method is not
// supported then an error is raised and execution is halted.
virtual boolean somFindMethodOk(::somId methodId, ::somMethodPtr* m);

// Returns 1 (true, if the indicated method is supported by this
// class and 0 (false) otherwise.
virtual boolean somSupportsMethod(::somId mId);

// The number of methods currently supported by this class,
// including inherited methods (both static and dynamic).
virtual long somGetNumMethods();

// The total size of an instance of the receiving class.
virtual long somGetInstanceSize();

// Returns the offset of instance data introduced by the receiver in
// an instance of the receiver. This method is obsolete and not useful in
// multiple-inheritance situations. The attribute somInstanceDataOffsets
// replaces this method.
virtual long somGetInstanceOffset();

// The size in bytes of the instance data introduced by the receiving
// class.
virtual long somGetInstancePartSize();

// Returns the index for the specified method. (A number that may
// change if any methods are added or deleted to this class object or
// any of its ancestors.) This number is the basis for other calls to
// get info about the method. Indexes start at 0. A -1 is returned if
// the method cannot be found.
virtual long somGetMethodIndex(::somId id);

// The number of static methods that this class has. Can
// be used by a child class when initializing its method table.
virtual long somGetNumStaticMethods();

```



```

// Returns the apply stub associated with the specified method,
// if one exists; otherwise NULL is returned. This method is obsolete,
// and retained for binary compatibility. In SOMr2, users never access
// apply stubs directly; the function somApply is used to invoke apply
// stubs. See somApply documentation for further information on apply
// stubs, and see somAddStaticMethod documentation for information
// on how apply stubs are registered by class implementations.
virtual ::somMethodPtr somGetApplyStub(::somId methodId);

// Finds the indicated method, which must be a static method supported
// by this class, and returns a pointer to its method procedure.
// If the method is not supported by the receiver (as a static method
// or at all) then a NULL pointer is returned.
virtual ::somMethodPtr somFindSMethod(::somId methodId);

// Uses <somFindSMethod>, and raises an error if the result is NULL.
virtual ::somMethodPtr somFindSMethodOk(::somId methodId);

// Returns the method descriptor for the indicated method. If
// this object does not support the indicated method then NULL is
// returned.
virtual ::somId somGetMethodDescriptor(::somId methodId);

// Returns the id of the <n>th method if one exists and NULL
// otherwise.
//
// The ordering of the methods is unpredictable, but will not change
// unless some change is made to the class or one of its ancestor classes.
//
// See CORBA documentation for info on method descriptors.
virtual ::somId somGetNthMethodInfo(long n, ::somId* descriptor);

// The class' pointer to the static <className>ClassData structure.
virtual void somSetClassData(::somClassDataStructure* cds);
virtual ::somClassDataStructure* somGetClassData();

// Equivalent to somNew except that somDefaultInit is not called.
virtual ::SOMObject* somNewNoInit();

// Equivalent to somRenew except that somDefaultInit is not called.
virtual ::SOMObject* somRenewNoInit(void* obj);

// A data token that identifies the introduced portion of this class
// within itself or any derived class. This token can be subsequently
// passed to the run-time somDataResolve function to locate the instance
// data introduced by this class in any object derived from this class.
virtual ::somDTToken somGetInstanceToken();

// Returns a data token that for the data member at offset
// "memberOffset" within the introduced portion of the class identified

```

```

// by instanceToken. The instance token must have been obtained from a
// previous invocation of somGetInstanceToken. The returned member
// token can be subsequently passed to the run-time somDataResolve
// function to locate the data member.
virtual ::somDToken somGetMemberToken(long memberOffset,
                                     ::somDToken instanceToken);

// If a method with id <methodId> is supported by the target class,
// the structure pointed to by <md> is loaded with method information
// and the value 1 is returned. In this case, if the method is not
// dynamic, md->method is loaded with the result of somClassResolve on
// the target class for the method. If the method is not supported,
// md->id is loaded with 0, and the value 0 is returned.
virtual boolean somGetMethodData(::somId methodId, ::somMethodData* md);

// Overrides the method table pointers to point to the redispatch stubs.
// All the methods except somDispatch methods are overridden.
virtual void somOverrideMtab();

// Returns the specified method's access token. This token can then
// be passed to method resolution routines, which use the token
// to select a method pointer from a method table.
virtual ::somMToken somGetMethodToken(::somId methodId);

// The parent classes of self.
virtual SOMClassSequence somGetParents();

// somInitMIClass implements the second phase of dynamic class creation:
// inheritance of interface and possibly implementation (instance
// variables) by suitable initialization of <self> (a class object).
//
// For somInitMIClass, the inherit_vars argument controls whether abstract
// or implementation inheritance is used. Inherit_vars is a 32 bit
// bit-vector. Implementation is inherited from parent i iff the bit
// 1<i is on, or i=32.
// On a class-by-class basis, for each class ancestor, implementation
// inheritance always takes precedence over abstract inheritance. This is
// necessary to guarantee that procedures supporting parent method calls
// (available on non-abstract parents) are always supported by parent
// instance variables.
//
// <className> is a string containing the class name. A copy is made, so
// the string may be freed upon return to the caller if this is desired.
//
// <parentClasses> is a SOMClassSequence containing pointers to the
// parent classes. somInitMIClass makes a copy of this, so it may
// be freed upon return to the caller if this is desired.
//
// <dataSize> is the space needed for the instance variables
// introduced by this class.
//
// <dataAlignment> specifies the desired byte alignment for instance

```

```

// data introduced by this class. A value of 0 selects a system-wide default;
// any other argument is taken as the desired byte alignment multiple. Thus,
// for example, even if a byte multiple of 8 is needed for double precision
// values on a given system (so 8 is the default), a class whose instance
data
// doesn't require this can indicate otherwise. If A is the next memory
// address available for holding instance data, the address that will be
// used is A + (A mod byte-alignment).
//
// <maxStaticMethods> is the maximum number of static methods that will be
// added to the initialized class using addStaticMethod.
//
// <majorVersion> indicates the major version number for this
// implementation of the class definition, and <minorVersion>
// indicates the minor version number.
virtual void somInitMClass(long inherit_vars, string className,
                           SOMClassSequence* parentClasses,
                           long dataSize, long dataAlignment,
                           long maxStaticMethods, long majorVersion,
                           long minorVersion);

// Returns the class' major and minor version numbers in the corresponding
// output parameters.
virtual void somGetVersionNumbers(long* majorVersion, long* minorVersion);

// Like <somFindSMMethodOK>, but without restriction to static methods.
virtual ::somMethodPtr somLookupMethod(::somId methodId);

// Equivalent to somRenew except that memory is not zeroed out.
virtual ::SOMObject* somRenewNoZero(void* obj);

// The purpose of this method is to load an object's method table.
// The SOM API requires that somRenewNoInitNoZero always be
// called when creating a new object whose metaclass is not SOMClass.
// This is because metaclasses must be guaranteed that they can use
// somRenewNoInitNoZero to track object creation if this is desired.
virtual ::SOMObject* somRenewNoInitNoZero(void* obj);

// Allocates memory to hold an object and returns a pointer to this memory.
// This is a nonstatic method, and cannot be overridden. The default
// implementation calls SOMMalloc, but a class designer can specify a
// different implementation using the somallocate modifier in IDL. The
// allocator takes the same parameters as this method.
::somToken somAllocate(long size);

// Zeros out the method table pointer stored in the word pointed to by
// memptr, and then deallocates the block of memory pointed to by memptr.
// This is a nonstatic method and cannot be overridden. The default
// deallocator called is SOMFree, but a class designer can specify a
// different deallocator using the somdeallocate modifier in IDL. The
// deallocator takes a char* (memptr) and a long (size) as arguments.
void somDeallocate(::somToken memptr);

// Returns a redispach stub for the indicated method, if possible.
// If not possible (because a valid redispach stub has not been

```

```

// registered, and there is insufficient information to dynamically
// construct one), then a NULL is returned.
virtual ::somMethodProc* somGetRdStub(::somId methodId);

// loads *md with the method data associated with the the nth method,
// or NULL if there is no such method. Returns true is successful;
// false otherwise.
virtual boolean somGetNthMethodData(long n, ::somMethodData* md);

// if the receiving class either introduces or overrides the
// indicated method, then its somMethodPtr is returned, otherwise
// NULL is returned.
virtual ::somMethodPtr somDefinedMethod(::somMToken method);

// uses namelookup from the target class to locate a method that has the
// indicated methodId, and returns that method's marshal plan if there
// is one. Otherwise, null is returned.
virtual ::somToken somGetMarshalPlan(::somId methodId);

// The alignment required for the instance data structure
// introduced by the receiving class.
long somDataAlignment;

#pragma SOMAttribute(somDataAlignment, readonly, virtualaccessors)

// The ancestors whose initializers the receiving
// class wants to directly invoke.
SOMClassSequence somDirectInitClasses;

#pragma SOMAttribute(somDirectInitClasses, readonly, virtualaccessors)
#pragma SOMReleaseOrder { \
    "somNew", \
    "somRenew", \
    "somInitClass", \
    "somClassReady", \
    "somGetName", \
    "somGetParent", \
    "somDescendedFrom", \
    "somCheckVersion", \
    "somFindMethod", \
    "somFindMethodOk", \
    "somSupportsMethod", \
    "somGetNumMethods", \
    "somGetInstanceSize", \
    "somGetInstanceOffset", \
    "somGetInstancePartSize", \
    "somGetMethodIndex", \
    "somGetNumStaticMethods", \
    *, \
    "somGetClassMtab", \
    "somAddStaticMethod", \
    "somOverrideSMethod", \

```

```

"somAddDynamicMethod", \
*, \
"somGetApplyStub", \
"somFindSMethod", \
"somFindSMethodOk", \
"somGetMethodDescriptor", \
"somGetNthMethodInfo", \
"somSetClassData", \
"somGetClassData", \
"somNewNoInit", \
"somRenewNoInit", \
"somGetInstanceToken", \
"somGetMemberToken", \
*, \
"somGetMethodData", \
"somOverrideMtab", \
"somGetMethodToken", \
"somGetParents", \
*, \
"somInitMIClass", \
"somGetVersionNumbers", \
"somLookupMethod", \
"_get_somInstanceDataOffsets", \
"somRenewNoZero", \
"somRenewNoInitNoZero", \
"somAllocate", \
"somDeallocate", \
"somGetRdStub", \
"somGetNthMethodData", \
*, \
*, \
"_get_somDirectInitClasses", \
*, \
*, \
*, \
*, \
*, \
*, \
*, \
*, \
"somDefinedMethod", \
*, \
*, \
*, \
*, \
*, \
*, \
*, \
*_get_somDataAlignment", \
*, \
*, \
*, \
*, \

```

```

        *, \
        "somGetMarshalPlan", \
        *)

};

#pragma SOMNonDTS(pop)
/* End SOMClass */
#endif /* _DTS_HH_INCLUDED_somcls */

```


Persistence SOM

The Persistence SOM (PSOM) Framework allows you to save objects so that they can exist after the process or program that created them has terminated. The SOM Persistence Framework provides support for transparently saving and restoring objects. By default, the saved data is stored in the file system, but you can subclass from the Persistence Framework to manage persistence through more specialized repositories. In addition, there are a variety of means of controlling when and how objects are saved.

This framework is replaced in SOMObjects 3.0 by the Persistent Object Service (also known as POSSOM). However, for those who may not have SOMObjects 3.0, this appendix will describe the PSOM Framework and illustrate how to define classes using PSOM so that they can be more easily migrated to POSSOM.

Persistent Objects: An Overview

This section presents a brief overview of the steps involved in creating, saving, and restoring a persistent object. In subsequent sections, we will examine the steps in more detail.

You make an object persistent, and save it, as follows:

1. *Instantiate an object of a type that derives from **SOMPPersistent-Object**.* Unlike DSOM, where any class type can be made into a

distributed object, you must explicitly specify that classes are capable of supporting persistent objects by deriving them from the class **SOMPPersistentObject**.

2. *Assign a persistent ID to the object.* The **SOMPPersistentObject** method is used to assign a persistent ID to the object. The ID itself is created through one of three methods, depending upon how you want to store the object.
3. *Save the object.* The object is saved through the **sompStoreObject** method of the **SOMPPersistentStorageMgr** class. Persistent objects are managed through an object of type **SOMPPersistentStorageMgr**. The persistent storage manager object controls the saving and restoring of objects and is the main interface to the Persistence Framework.

Once the object has been saved, you can restore it as follows:

1. *Instantiate an object of the same type as the saved object.*
2. *Assign the persistent ID to the object.* The persistent ID must be the same as that of the original saved object. There are a number of ways to re-create/save the original ID.
3. *Restore the object.* The object is restored through the **sompRestoreObject** method of the **SOMPPersistentStorageMgr** class.

Defining a Persistent-Capable Class

Any object that is to be managed through the Persistence Framework must be of a class that is derived from **SOMPPersistentObject**. Each data member that is to be saved with the object must be made into an attribute and given the IDL persistent modifier. This is shown next for the class **PCount**.

The class **PCount** derives directly from **SOMPPersistentObject** at line 3. **PCount** has only a single data member, **count**, which is made an attribute through the **SOMAttribute** pragma at line 9, and is made a persistent attribute through the **SOMIDLPass** pragma at line 10. The **SOMIDLPass** pragma adds the string "count: persistent;" to the beginning of the implementation section in the generated IDL. This IDL modifier indicates that the attribute **count** is persistent and should be saved and restored through the Persistence Framework when objects of class **PCount** are saved and restored. The DTS C++ compiler already generates a modifier statement for the data member, but you may have multiple modifier statements for a given component, so we can simply add the extra information as a single statement.

Definition of DirectToSOM C++ Class PCount (pcount\pcount.hh):

```

1  #include <somp.hh>
2
3  class PCount : public SOMPPersistentObject {
4      #pragma SOMClassName(*, "PCount")

```

```

5      #pragma SOMIDLPass(*, "Implementation-End", \
6          "dllname = \"pcount.dll\";")
7      public:
8          PCount() { count = 0; }
9          int count;
10         #pragma SOMAttribute(count)
11         #pragma SOMIDLPass(*, "Implementation-Begin", \
12             "count: persistent;")
13     };

```

Assigning a Persistent ID

The data for a persistent object is stored by default in the file system. Using the default Persistence Framework, the persistent ID identifies the file where the data is stored, how the data is stored (binary or ASCII), and the location within the file where the data resides. There are three ways to assign a persistent ID to an object:

1. *Independent system-generated ID.* The ID is created by the system based on the next available ID number. Nothing else is taken into account. The ID is assigned to the object using the **SOMPersistentObject** method **sompInitNextAvail**.
2. *Dependent system-generated ID.* The ID is generated by the system so that the new object will be stored “near” another object. The ID is assigned to the object using the **SOMPersistentObject** method **sompInitNearObject**.
3. *User-defined ID.* The ID is specified by the user. For the default Persistence Framework, this involves specifying the storage type (ASCII or binary), the file and the offset where the object is to be stored. The ID is assigned to the object using the **SOMPersistentObject** method **sompInitGivenId**.

A common way to work with persistent IDs is to create a user-defined or independent system-generated ID for the initial object, then create dependent system-generated IDs for the remainder of the objects using the initial ID.

Assigning an Independent System-Generated ID with **sompInitNextAvail**

Independent system-generated IDs are created through the **SOMPidAssigner** class. An instance of this class is used to manage the assignment of IDs. To use this approach, you simply instantiate an object of type **SOMPidAssigner**, then invoke the method **sompInitNextAvail** against the persistent object, passing the address of a **SOMPidAssigner** instance.

As our first example of working with persistent objects, the following shows a simple instance of creating and saving a persistent object by assign-

ing an independent system-generated ID. The implementation file for the class `PCount` contains only the **SOMInitModule** function and a **SOMDefine** for the class. **SOMDefine** is necessary to force the compiler to generate the SOM class data structures for the class, since `PCount` has no out-of-line member functions.

In the client program, the variable `psm` is defined at line 11 as an instance of the persistent storage manager class **SOMPPersistentStorageMgr**. The persistent storage manager will be used to save the persistent object. Next, at line 14, is the declaration of an object `pobj`, of type `PCount`, which is the class defined earlier. This will be the persistent object, whose data will be saved by the persistent storage manager `psm`. Then the function `AssignPersistentID` is called at line 17 to assign an ID to the object `pobj`. Once the ID has been assigned, the persistent storage manager method **sompStoreObject** is invoked against object `psm` at line 20 to save the object `pobj`.

The function `AssignPersistentId` at lines 26 through 38 assigns a persistent ID to an object using the next available persistent ID. This is achieved by declaring an object ID assigner of type **SOMPIdAssigner**, which manages persistent IDs. The method **sompInitNextAvail** is called, which uses the assigner object to determine the next ID to assign.

Implementation of DirectToSOM C++ Class PCount (pcount\pcount.cpp):

```

1  #include <fstream.h>
2  #include "pcount.hh"
3
4  // No out-of-line member functions, so use SOMDefine
5  // to create SOM class data structures
6  #pragma SOMDefine(PCount)
7
8  SOMEXTERN void SOMLINK EXPORT SOMInitModule(
9      long major, long minor, string className)
10 {
11     PCountNewClass(major, minor);
12 }
```

Client of DirectToSOM C++ Class PCount (pcount\tst1.cpp):

```

1  #include <iostream.h>
2  #include "pcount.hh"
3
4  int AssignPersistentId(PCount&);
5
6  int main(int argc, char *argv[])
7  {
8      __SOMEnv = SOM_CreateLocalEnvironment();
9
10     // instantiate a persistent storage manager object
11     static SOMPPersistentStorageMgr psm;
12
13     // instantiate the persistent object
```

```

14     PCount pobj;
15
16     // assign persistent id to the object
17     AssignPersistentId(pobj);
18
19     // save object through persistent storage manager
20     psm.sompStoreObject(&pobj);
21
22     SOM_DestroyLocalEnvironment(__SOMEnv);
23 }
24
25
26 int AssignPersistentId(PCount &pobj)
27 {
28     // instantiate an id assigner object
29     static SOMPidAssigner assigner;
30     char id[SOMPMAXIDSIZE];
31
32     // assign the next available persistent id
33     pobj.sompInitNextAvail(&assigner);
34
35     // display the persistent id
36     pobj.sompGetPersistentIdString(id);
37     cout << "Object ID is " << id << endl;
38     return TRUE;
39 }

```

The Persistent ID

If you compile and run the previous example, you will get output such as the following:

```
Object ID is SOMPASCII:.\p0000000;0
```

This is the assigned object ID for the persistent object. Using the default Persistence Framework, an object ID consists of three parts:

storage_format:storage_directory\filename:offset

The storage format will be one of **SOMPASCII** or **SOMPBinary**, depending upon how you want the object to be saved. The storage directory is where the persistent storage file is located; the file name is the name of the file to be used to save the object; and the offset is the location within the file where the object is stored. (Note that these examples were run in OS/2. If you are using AIX, the directory separator character is / instead of \.)

The real SOM terminology for these components is:

<IOGroupMgrClassName>:<IOGroupName>:<IOGroupOffset>

where an IO group is a more generic concept used to reflect groupings of saved objects, not necessarily in a file system. This abstraction is useful if

you want to customize the Persistence Framework, but is more confusing than helpful for the scope of this discussion.

When you use a **SOMPIdAssigner** object to generate independent persistent IDs, it always uses a storage format of **SOMPASCII** and an offset of 0. The storage directory is set to the value of the **SOMP_PERSIST** environment variable. If **SOMP_PERSIST** is not set, the default is the current directory.

The file name is set to Pxxx, where xxx is the number in hex of the next available ID number. The ID assigner saves the most recently used ID in the file **somplast.id** in the storage directory. When it needs a new ID, it looks for this file and, if found, reads the number, and assigns the hex value as the ID number. If the file is not found, 0 is assumed. The file is then updated/created with the next available ID number, obtained by incrementing the current number.

Let's reexamine the persistent ID displayed by the previous test program:

```
Object ID is SOMPASCII:.\p00000000:0
```

The storage format is ASCII and the offset is 0. The storage directory is the current directory, and the storage file name is p00000000. If you examine the current directory after running the program, you will see the following new files:

```
somplast.id
p00000000
```

somplast.id will contain the following line:

1 is the next available ID

and p00000000 will contain the saved object, which will look something like this:

```
SOMPascii OTOC          06
6 PCount0 0 22 SOMPAttrEncoderDecoder5 count4 0 7 $_EOA_$
Number of Persistent Objects:          1
Last assigned offset:          0
Num      flag          start      len
-----
          0          0          29          57
```

If you ran the program several times, the value in **somplast.id** would increment by one each time, and a new file Pxxx would be created for each execution, corresponding to the next available ID number.

Assigning a User-Defined ID

User-defined IDs are created through an instance of the **SOMPPersistentId** class. This class provides a method **somutSetIdString** that allows you to assign an ID string to a persistent ID instance. Then you can supply this instance as a parameter to the **SOMPPersistentObject** method **sompInitGivenId** to assign the user-defined ID to the persistent object.

In the following code segment, I replaced the **AssignPersistentId** function from the previous example with one that sets the persistent ID from a user-defined ID. The variable **pid** is an instance of the persistent ID class **SOMPPersistentId**. The file name for the persistent data store is set to **pfile.dat** by invoking the method **sompSetIOGroupName** against the persistent ID object **pid**. By default, the storage format and offset will default to **SOMPASCII** and 0 respectively.

Assigning a User-Defined ID (*pcount\tst2.cpp*):

```
int AssignPersistentId(PCount &pobj)
{
    // instantiate an persistent id object
    static SOMPPersistentId pid;
    char id[SOMPWXIDSIZE];

    // set the file name,
    // format defaults to ASCII, offset to 0
    pid.sompSetIOGroupName("pfile.dat");

    pobj.sompInitGivenId(&pid);

    // display the persistent id
    pobj.sompGetPersistentIdString(id);
    cout << "Object ID is " << id << endl;
    return TRUE;
}
```

You can set the group explicitly using the **sompSetIOGroupMgrClassName** and **sompSetGroupOffset** methods, or you can set all three at once using the **somutSetIdString** method. For example, the following creates a persistent ID with the storage format of binary, in the file **\jennifer\psom\psom.out**, at offset 1:

Assigning a Group and Offset for a User-Defined ID (*pcount\tst3.cpp*):

```
int AssignPersistentId(PCount &pobj)
{
    // instantiate an persistent id object
    static SOMPPersistentId pid;
    char id[SOMPWXIDSIZE];

    // set the format to Binary,
    // file name to psom.out, and offset to 1
```

Continued

```

pid.somutSetIdString(
    "SOMPBinary:\\jennifer\\psom\\psom.out:1");

pobj.sompInitGivenId(&pid);

// display the persistent id
pobj.sompGetPersistentIdString(id);
cout << "Object ID is " << id << endl;
return TRUE;
}

```

Dependent System-Generated ID

So far we've looked at two of the three methods for assigning a persistent ID to an object: independent system-generated and user-defined. The third way is through a dependent system-generated ID, where the Persistence Framework generates an ID such that the object will be stored near another ID. In the context of the default framework, this simply means the new persistent object should be stored in the same file as the existing object, at the next available offset.

The following programming example shows how to create a persistent ID near another persistent ID. The program declares two `PCount` objects, `pobj1` and `pobj2`. `pobj1` is assigned a user-defined persistent ID through `AssignPersistentId`, and `pobj2` is assigned an ID near `pobj1` through the **SOMPPersistentObject** method `sompInitNearObject`. Then both objects are saved in the persistent store through `sompStoreObject`. If the program were run in OS/2, the result would be:

```
Object ID is SOMPASCII:pfile.dat:0
```

```
Object ID is SOMPASCII:pfile.dat:1
```

Note that the first object ID is at offset 0 in file `pfile.dat`, while the second, system-generated, ID is at offset 1 in the same file.

Assigning a Dependent System-Generated ID (pcount\tst4.cpp):

```

1  #include <iostream.h>
2  #include "pcount.hh"
3
4  int AssignPersistentId(PCount&);
5  void DisplayPersistentId(PCount &pobj);
6
7  int main(int argc, char *argv[])
8  {
9      __SOMEnv = SOM_CreateLocalEnvironment();
10
11     // instantiate a persistent storage manager object
12     static SOMPPersistentStorageMgr psm;

```



```

13
14 // instantiate the persistent objects
15 PCount pobj1, pobj2;
16
17 // assign persistent id to pobj1
18 AssignPersistentId(pobj1);
19
20 // assign persistent id near pobj1
21 pobj2.sompInitNearObject(&pobj1);
22 DisplayPersistentId(pobj2);
23
24 // save objects through persistent storage manager
25 psm.sompStoreObject(&pobj1);
26 psm.sompStoreObject(&pobj2);
27
28 SOM_DestroyLocalEnvironment(__SOMEnv);
29 }
30
31
32 int AssignPersistentId(PCount &pobj)
33 {
34     // instantiate an persistent id object
35     static SOMPPersistentId pid;
36
37     // set the file name,
38     // format defaults to ASCII, offset to 0
39     pid.sompSetIOGroupName("pfile.dat");
40
41     pobj.sompInitGivenId(&pid);
42     DisplayPersistentId(pobj);
43     return TRUE;
44 }
45
46
47 void DisplayPersistentId(PCount &pobj)
48 {
49     char id[SOMPMAXIDSIZE];
50
51     pobj.sompGetPersistentIdString(id);
52     cout << "Object ID is " << id << endl;
53 }

```

Restoring Persistent Objects

So far, all I've done is explain how to create and save persistent objects. This is not particularly useful unless you can actually restore them, too. Objects are restored, as they are saved, through the persistent storage manager object, in this case using the method **sompRestoreObject**.

The process is simple. You must first create a persistent ID that matches that of the object you want to restore. Then, you ask the persistent

storage manager to restore that object from the persistent store by supplying the persistent ID. The Persistence Framework restores an object first by creating an object of the appropriate type through **somNew**, which calls **somDefaultInit** for the class, which maps to the C++ default constructor. Then, for each persistent attribute to be restored, the **_set** attribute method is called to assign the restored value.

The following example shows how to restore an object from the persistent datastore. Rather than instantiate an automatic object of type **PCount** and assign it an ID through **AssignPersistentId**, in this example, the function **GetPersistentObject** is called, which will return a pointer to an object of type **PCount**. The object returned will either be an object restored from the datastore or a new object. In either case, the program increments the object count and displays it. Then the object is saved to the datastore (this may be an update if the object already exists), and the storage for that object is deleted. Each time the program is called, count increments by 1, starting with an initial value of 0, as set in the constructor for **PCount**.

GetPersistentObject first creates a persistent ID (using the same steps as in **AssignPersistentId**, previously). Next, the **SOMPPersistentStorageMgr** method **sompObjectExists** is called with this persistent ID to determine whether the object already exists in the persistent datastore. If it does, **sompRestoreObject** is called to retrieve it; if not, a new one is allocated and assigned the persistent ID pid.

sompRestoreObject takes the address of a persistent ID, allocates storage of the correct type, and restores the data corresponding to that persistent ID into the allocated object. It assigns the persistent ID to that restored object and returns its address.

Restoring an Object (pcount\ tst5.cpp):

```

1  #include <iostream.h>
2  #include "pcount.hh"
3
4  PCount *GetPersistentObject();
5
6  int main(int argc, char *argv[])
7  {
8      __SOMEnv = SOM_CreateLocalEnvironment();
9
10     // instantiate a persistent storage manager object
11     static SOMPPersistentStorageMgr psm;
12
13     // instantiate the persistent object
14     PCount *pobj = GetPersistentObject();
15
16     ++pobj->count;
17     cout << "Count is " << pobj->count << endl;
18
19     // save object through persistent storage manager
20     psm.sompStoreObject(pobj);

```

```

21
22     // free the object
23     delete pObj;
24
25     SOM_DestroyLocalEnvironment(__SOMEnv);
26 }
27
28
29 PCount *GetPersistentObject()
30 {
31     static SOMPPersistentStorageMgr psm;
32     static SOMPPersistentId pid;
33     PCount *pobj;
34
35     // set the file name,
36     // format defaults to ASCII, offset to 0
37     pid.sompSetIOGroupName("pfile.dat");
38
39     // if persistent object exists, restore it
40     if (psm.sompObjectExists(&pid)) {
41         pObj = (PCount *)psm.sompRestoreObject(&pid);
42     } else {
43         pObj = new PCount;
44         pObj->sompInitGivenId(&pid);
45     }
46     return pObj;
47 }

```

Keep in mind that you must first restore an object in order to rewrite it back to the persistent datastore. The storage manager always writes a new value to the end of the datastore. If an object that has been restored is rewritten, the original value is marked as unused. For example, the datastore after running the previous example several iterations under OS/2 (note that the storage format may change from system to system and release to release) is shown next. With this storage format, each string ending with `$_EOA_$` represents a single saved value for the `PCount` object. The line at the bottom of the file under `start` indicates where object 0 starts in the line. Each time you run the program, a new object value is added to the list, and the value under `start` increments to indicate where the current value actually is stored.

If you don't restore the object first, the storage manager will not mark the original value as unused (in this example, by incrementing the `start` value), so when you try to restore the object, you will get the original value (given by `start`), not the new value. However, because of this behavior of writing a new value to the end of the datastore, the datastore will eventually grow large and you will need to compact it. See *Compacting the Datastore* later in this appendix for details.

```

SOMPAscii OTOC                200
6 PCount0 0 22 SOMPAttrEncoderDecoder5 count4 1 7 $_EOA_$ PCount0 0 22
SOMPAttrEncoderDecoder5 count4 2 7 $_EOA_$ PCount0 0 22
SOMPAttrEncoderDecoder5 count4 3 7 $_EOA_$

```

Continued

Number of Persistent Objects:				1
Last assigned offset:				0
Num	flag	start	len	

	0	0	143	57

Working with Persistent Objects

Deleting a Persistent Object

A persistent object has the feature that it is, well, persistent! So, when your program or process ends, the data doesn't go away. But you will eventually need to delete a persistent object permanently, and to do this you use the **SOMPPersistentStorageMgr** method **sompDeleteObject**. In the following example, the object is deleted from the persistent datastore when the count exceeds 3. **sompDeleteObject** will also delete the in-memory copy of the object, so there is no need to explicitly delete it (in fact, if you do, you will get an exception).

Deleting a Persistent Object (pcount\tst6.cpp):

```

1  #include <iostream.h>
2  #include "pcount.hh"
3
4  PCount *GetPersistentObject();
5
6  int main(int argc, char *argv[])
7  {
8      __SOMEnv = SOM_CreateLocalEnvironment();
9
10     // instantiate a persistent storage manager object
11     static SOMPPersistentStorageMgr psm;
12
13     // instantiate the persistent object
14     PCount *pobj = GetPersistentObject();
15
16     ++pobj->count;
17     cout << "Count is " << pobj->count << endl;
18
19     if (pobj->count > 3) {
20         // delete the object, also deletes pobj
21         psm.sompDeleteObject(pobj->sompGetPersistentId());
22     } else {
23         // save object through persistent storage manager
24         psm.sompStoreObject(pobj);
25         delete pobj;
26     }
27
28     SOM_DestroyLocalEnvironment(__SOMEnv);

```

```

29  }
30
31
32  PCount *GetPersistentObject()
33  {
34      static SOMPPersistentStorageMgr psm;
35      static SOMPPersistentId pid;
36      PCount *pobj;
37
38      // set the file name,
39      // format defaults to ASCII, offset to 0
40      pid.sompSetIOGroupName("pfile.dat");
41
42      // if persistent object exists, restore it
43      if (psm.sompObjectExists(&pid)) {
44          pobj = (PCount *)psm.sompRestoreObject(&pid);
45      } else {
46          pobj = new PCount;
47          pobj->sompInitGivenId(&pid);
48      }
49      return pobj;
50  }

```

Marking an Object as Dirty

In all of the examples so far, whenever we have requested that an object be saved, the persistent storage manager always wrote that object out to the datastore. But what if the object hasn't changed? It would be a waste to write it out. In these simple examples, it would not be a problem, but as you start dealing with a lot more objects, it becomes a concern.

You can control this behavior through the **SOMPPersistentObject** method **sompIsDirty**. The Persistence Framework invokes this method against an object to be saved to determine whether it actually needs to be written out. If **sompIsDirty** returns true, the object is written to the datastore; otherwise, it is not written out. The default version of **sompIsDirty** supplied by **SOMPPersistentObject** always returns true, but you can override it to optimize the I/O performed by the framework.

In the example with class **PCount**, we would only want to mark an object to be saved if the count had been updated. The count will only be updated outside the class through the **_set_count** method (since **count** is an attribute). Therefore, we can supply a user-defined version of the **_set_count** method and indicate that the object should be saved when the count has been updated.

The next example shows the updated version of the class description to support the **sompIsDirty** method for class **PCount**. I added an override of **sompIsDirty** and changed the attribute description for **count** to specify **noset**. This means that I will be supplying the **_set_count** routine rather than having the compiler supply one for me.

In the updated version of the class implementation, I added two new methods: `_set_count` and `sompIsDirty`; `_set_count` sets the “dirty” flag for the object by invoking `sompSetDirty` against itself; `sompIsDirty` simply returns the “dirty” flag setting for the current object. Now when the Persistence Framework needs to determine whether to save a `PCount` object, invoking `sompIsDirty` will indicate that. Note that the framework will reset the “dirty” flag once an object has been saved. (If you are using VisualAge C++ for OS/2, version 3, you will need to compile this new version of `PCount` with `-yxqnosomvolattr`. See the discussion of the `SOMAttribute` program in Chapter 4.)

PCount Class Supporting Dirty Flag (pcount/pcount2.hh):

```

1  #include <somp.hh>
2
3  class PCount : public SOMPersistentObject {
4      #pragma SOMClassName(*, "PCount")
5      #pragma SOMIDLPass(*, "Implementation-End", \
6          "dllname = \"pcount.dll\";")
7  public:
8      PCount() { count = 0; }
9      boolean sompIsDirty();
10     int count;
11     #pragma SOMAttribute(count, noset)
12     #pragma SOMIDLPass(*, "Implementation-Begin", \
13         "count: persistent;")
14 };
15
```

PCount Implementation Supporting Dirty Flag (pcount/pcount2.cpp):

```

1  #include <iostream.h>
2  #include "pcount.hh"
3
4  void PCount::_set_count(int newValue)
5  {
6      count = newValue;
7      // set the dirty bit for the object
8      this->sompSetDirty();
9  }
10
11 boolean PCount::sompIsDirty()
12 {
13     // return the dirty bit for the object
14     return this->sompGetDirty();
15 }
16
17
18 SOMEXTERN void SOMLINK EXPORT SOMInitModule(
19     long major, long minor, string className)
20 {

```

```

21     PCountNewClass(major, minor);
22 }

```

Recursive Save and Restore

When objects are saved and restored with **sompStoreObject** and **sompRestoreObject**, the Persistence Framework automatically saves and restores all children objects recursively. There may be situations, however, when you want to save or restore only a portion of an object hierarchy. In this case, you can use the **sompStoreObjectWithoutChildren** and the **sompRestoreObjectWithoutChildren** methods. The first saves only the parent object to the datastore; the second allocates storage and assigns the persistent IDs to both the parent object and any of its children, but only actually restores the data into the parent object.

Compacting the Datastore

As discussed earlier, the storage manager always writes a new value to the end of the datastore. If this is a rewrite of an object that has been restored, the original value is marked as unused. Because of this behavior of writing a new value to the end of the datastore, it will eventually grow large and you will need to compact it. There are two ways to do this, either through the **SOMPPersistentObject** method **sompMarkForCompaction** or the utility **srgarbel** (short for garbage collection).

When applied to a persistent object, **sompMarkForCompaction** indicates to the persistent storage manager that the next time this object is stored, all data within the file should be compacted. For the default framework, the **srgarbel** utility takes the following parameters:

```
srgarbel [-i<SOMPASCII|SOMPBinary>] <data_store>
```

where the default storage format is **SOMPASCII**.

Persistent Message Queue

In the following example, I rewrote the Chapter 3 version of the message queue example to make the queues persistent. The overall design is that when the program terminates, all changed message queues will be written to the persistent datastore, one queue per file, through the **MessageQueueServer** destructor. Then, when the program requests a message queue through **MessageQueueServer::GetMessageQueue(char *)**, the queue will be restored from the persistent store if found; otherwise, a new message queue will be created. When a change is made to a message queue, the dirty flag is set for that queue, so that a message queue will be saved only if it has changed.

The `MessageQueue` and the `MessageQueue::Mqueue` classes in the `mqueue.hh` file both now inherit from **SOMPPersistentObject**. This is because attributes of both classes will be saved. For the `MessageQueue` class, the data member `mq`, `last`, `name`, and `count` are declared as persistent attributes. For the `MessageQueue::Mqueue` class, the persistent attributes are the data members `next` and `message`. Because the Persistence Framework stores objects recursively, when an object of type `MessageQueue` is saved, the entire underlying message queue will be saved with that ID. When a `MessageQueue::Mqueue` object is saved, the `MessageQueue::Mqueue` object corresponding to that object's `mq` attribute will be saved. The `next` pointer for this object will be followed and saved, and so on, until the entire list is recursively saved.

Except for the addition of the **sompPassivate**, **sompActivated**, and **sompIsDirty** methods, the new version of the `mqueue.cpp` file is not much changed from the nonpersistent version. The only changes occur in the `MessageQueue` `Send`, `Receive`, and `Clear` routines. Whenever a new `MessageQueue::Mqueue` object is created, it must be assigned a persistent ID, so that it can be saved and restored with the message queue itself. To achieve this, `MessageQueue::Send` invokes **sompInitNearObject** against the new object at line 74, passing the message queue as the reference object. This will assign a persistent ID that is in the same datastore file as the `MessageQueue` itself.

In addition, because each of the methods `Send`, `Receive`, and `Clear` update the message queue, the method **sompSetDirty** is invoked to indicate that the queue should be saved. Note that the **sompIsDirty** method is also called for the first element in the queue of messages. This ensures that the underlying queue is saved also, but only when the containing `MessageQueue` object is modified.

The **sompPassivate** and **sompActivated** methods are called when an object is stored and restored respectively. You can use them to perform specific actions when these events take place. I am using them in this example just to trace the saving and restoring of the message queues.

The `mqserver.h` file is the same as before, but the `mqserver.cpp` file has been updated to save and restore the queues from the persistent datastore. The constructor for the `MessageQueueServer` class has not changed, but the destructor now saves each queue to the persistent datastore through the **sompStoreObject** method. In `GetMessageQueue(char *)`, if the requested message queue is found in the in-memory array, it is returned, as before. But now, if the queue is not found, it will be restored from the persistent datastore if it exists there; otherwise, a new message queue will be created. The persistent ID for a message queue is the queue name with `.pst` appended.

In the makefile, I linked the `sommem1.obj` binary discussed in Chapter 8 into both `mqueue.dll` and `tstmq.exe`. The calls to **new** and **delete** to allocate and delete storage for the message text and message queue name will therefore use the SOM storage allocation routines rather than the standard C++

routines to avoid problems with mixing the compiler and the SOM memory management routines. You cannot use the compiler `delete` operator to deallocate memory that was allocated through **SOMMalloc**, and vice-versa.

If the standard C++ **new** operator were used in the constructors to allocate the message text and queue name storage, **delete** would work fine for deleting such messages. But, when an object is restored from the persistent datastore, SOM uses **SOMMalloc** to allocate storage for these attributes, in which case **delete** would not be valid to use, because the storage was allocated using a different storage manager.

Note that the **-p** SOM compiler option is used in the makefile when updating the information repository to ensure that private class information is incorporated. If this option were not included, the persistent attribute information would be ignored, and private class information would not be saved.

Definition of DirectToSOM C++ Class MessageQueue (mqueue/mqueue.hh):

```

1  #ifndef MQUEUE_H
2  #define MQUEUE_H
3
4  #include <somp.hh>
5
6  #define SUCCESS 0
7  #define FAIL 1
8  #define MAX_QUEUE_NAME_LEN 20
9  #define MAX_MESSAGE_LEN 256
10
11 class MessageQueue :
12     public SOMPPersistentObject {
13     #pragma SOMClassName(*, "MessageQueue")
14     #pragma SOMIDLPass(*, "Implementation-End", \
15         *dllname = \"mqueue.dll\";*)
16
17     struct Mqueue : public SOMPPersistentObject {
18         Mqueue *next;
19         char* message;
20         Mqueue();
21         Mqueue(char *);
22         ~Mqueue();
23         void sompPassivate();
24         void sompActivated();
25         boolean sompIsDirty();
26         #pragma SOMAttribute(next)
27         #pragma SOMIDLPass(*, "Implementation-Begin", \
28             *next: persistent;*)
29         #pragma SOMAttribute(message)
30         #pragma SOMIDLPass(*, "Implementation-Begin", \
31             *message: persistent;*)
32     };
33     Mqueue *mq, *last;
34     int count;

```

Continued

```

35     public:
36         char *name;
37         MessageQueue();
38         MessageQueue(char *);
39         ~MessageQueue();
40         void Clear();
41         int Send(char *);
42         int Receive(char *);
43         void Dump();
44         int Count();
45         boolean sompIsDirty();
46         void sompPassivate();
47         void sompActivated();
48         #pragma SOMAttribute(mq)
49         #pragma SOMIDLPass(*, "Implementation-Begin", \
50             "mq: persistent;")
51         #pragma SOMAttribute(last)
52         #pragma SOMIDLPass(*, "Implementation-Begin", \
53             "last: persistent;")
54         #pragma SOMAttribute(count)
55         #pragma SOMIDLPass(*, "Implementation-Begin", \
56             "count: persistent;")
57         #pragma SOMAttribute(name)
58         #pragma SOMIDLPass(*, "Implementation-Begin", \
59             "name: persistent;")
60     };
61
62 #endif

```

Definition of Native C++ Class MessageQueueManager (mqueue/mqmgr.h):

```

1  #ifndef MQSERVER_H
2  #define MQSERVER_H
3
4  #include "mqueue.hh"
5
6  #define MAX_QUEUES 20
7
8  class MessageQueueManager {
9      MessageQueue *mqueues[MAX_QUEUES];
10  public:
11      MessageQueueManager();
12      ~MessageQueueManager();
13      MessageQueue *GetMessageQueue(char *);
14      MessageQueue *GetMessageQueue(int);
15  };
16
17 #endif

```

Implementation of DirectToSOM C++ Class MessageQueue (mqueue/mqueue.cpp):

```

1  #include <iostream.h>
2  #include <assert.h>

```

```

3  #include "mqueue.hh"
4
5  // Need a no-argument/default constructor
6  // for PSOM to call
7  MessageQueue::Mqueue::Mqueue()
8  {
9      next = NULL;
10     message = NULL;
11 }
12
13 MessageQueue::Mqueue::Mqueue(char *elemMessage)
14 {
15     next = NULL;
16     message = new char[strlen(elemMessage) + 1];
17     assert(message != NULL);
18     strcpy(message, elemMessage);
19 }
20
21 MessageQueue::Mqueue::~Mqueue()
22 {
23     delete message;
24     if (next)
25         delete next;
26 }
27
28 void MessageQueue::Mqueue::sompPassivate()
29 {
30     cout << "\tSaving message elem: "
31           << message << endl;
32 }
33
34 void MessageQueue::Mqueue::sompActivated()
35 {
36     cout << "\tRestored message elem: "
37           << message << endl;
38 }
39
40 boolean MessageQueue::Mqueue::sompIsDirty()
41 {
42     return this->sompGetDirty();
43 }
44
45 MessageQueue::MessageQueue()
46 {
47     name = NULL;
48     last = mq = NULL;
49     count = 0;
50 }
51
52 MessageQueue::MessageQueue(char *qname)
53 {
54     last = mq = NULL;
55     count = 0;
56     name = new char[strlen(qname) + 1];
57     assert(name != NULL);

```

Continued

```

58     strcpy(name, qname);
59 }
60
61 MessageQueue::~MessageQueue()
62 {
63     Clear();
64     if (name)
65         delete name;
66 }
67
68 int MessageQueue::Send(char *message)
69 {
70     Mqueue *elem;
71     if (! (elem = new Mqueue(message)))
72         return FAIL;
73     // assign persistent id near owner queue
74     elem->sompInitNearObject(this);
75     if (mq == NULL) {
76         mq = last = elem;
77     } else {
78         last->next = elem;
79         last = elem;
80     }
81     ++count;
82     sompSetDirty();
83     // only save when message queue changes
84     mq->sompSetDirty();
85     return SUCCESS;
86 }
87
88 int MessageQueue::Receive(char *buf)
89 {
90     if (!mq) {
91         return FAIL;
92     }
93     Mqueue *elem = mq;
94     mq = mq->next;
95     --count;
96     if (last == elem)
97         last = NULL;
98     strcpy(buf, elem->message);
99     // so don't delete entire chain
100    elem->next = NULL;
101    delete elem;
102    sompSetDirty();
103    if (mq)
104        // only save when message queue changes
105        mq->sompSetDirty();
106    return SUCCESS;
107 }
108
109 int MessageQueue::Count()
110 {
111     return count;

```

```

112 }
113
114 void MessageQueue::Dump()
115 {
116     int i = 1;
117     cout << "Dumping queue " << name << endl;
118     for (Mqueue *cur = mq; cur != NULL;
119          cur=cur->next, i++)
120         cout << '\t' << i << " : "
121             << cur->message << endl;
122 }
123
124 void MessageQueue::Clear()
125 {
126     if (mq != NULL) {
127         delete mq;
128         mq = last = NULL;
129     }
130     count = 0;
131     sompSetDirty();
132 }
133
134 void MessageQueue::sompPassivate()
135 {
136     cout << "Saving queue: " << name << endl;
137 }
138
139 void MessageQueue::sompActivated()
140 {
141     cout << "Restored queue: " << name << endl;
142 }
143
144 boolean MessageQueue::sompIsDirty()
145 {
146     return this->sompGetDirty();
147 }
148
149 SOMEXTERN void SOMLINK EXPORT SOMInitModule(
150     long major, long minor, string className)
151 {
152     MessageQueueNewClass(major, minor);
153 }

```

**Implementation of DirectToSOM C++ Class MessageQueueManager
(mqmgr.cpp):**

```

1  #include <iostream.h>
2  #include <assert.h>
3  #include "mqmgr.h"
4
5  #define MAX_QUEUES 20
6
7  MessageQueueManager::MessageQueueManager()

```

Continued

```

8  {
9      for (int i=0; i<MAX_QUEUES; i++)
10         mqueues[i] = NULL;
11  }
12
13  MessageQueueManager::~MessageQueueManager()
14  {
15      static SOMPPersistentStorageMgr psm;
16
17      for (int i=0; i<MAX_QUEUES; i++)
18         if (mqueues[i]) {
19             psm.sompStoreObject(mqueues[i]);
20             delete mqueues[i];
21         }
22  }
23
24  MessageQueue *MessageQueueManager::
25      GetMessageQueue(char *name)
26  {
27      for (int i=0; i<MAX_QUEUES; i++)
28         if (mqueues[i] && mqueues[i]->name &&
29             strcmp(mqueues[i]->name, name) == 0)
30             return mqueues[i];
31      for (i=0; i<MAX_QUEUES && mqueues[i]; i++)
32          ;
33      if (i == MAX_QUEUES)
34          return NULL;
35
36      // Look for the list in persistent storage
37      static SOMPPersistentStorageMgr psm;
38      static SOMPPersistentId pid;
39
40      char *format = "SOMPAscii:%s.pst:0";
41      char *pstring =
42          new char[strlen(name) + strlen(format)];
43      assert(pstring != NULL);
44      sprintf(pstring, format, name);
45      pid.somutSetIdString(pstring);
46      delete pstring;
47
48      // if persistent object exists, restore it
49      if (psm.sompObjectExists(&pid)) {
50          mqueues[i] = (MessageQueue *)
51              psm.sompRestoreObject(&pid);
52      } else {
53          if (! (mqueues[i] = new MessageQueue(name)))
54              return NULL;
55          mqueues[i]->somplnitGivenId(&pid);
56      }
57      return mqueues[i];
58  }
59
60  MessageQueue *MessageQueueManager::

```

```

61         GetMessageQueue(int qnum)
62     {
63         if (qnum < MAX_QUEUES && mqueues[qnum])
64             return(mqueues[qnum]);
65         return NULL;
66     }

```

Client of DirectToSOM C++ Class MessageQueue (mqueue/tstmq.cpp):

```

1  #include <iostream.h>
2  #include "mqmgr.h"
3
4  int main(int argc, char *argv[])
5  {
6      MessageQueueManager mqlist;
7      __SOMEnv = SOM_CreateLocalEnvironment();
8
9      for ( ;; ) {
10         char choice, qname[MAX_QUEUE_NAME_LEN],
11             message[MAX_MESSAGE_LEN];
12         MessageQueue *mq;
13         cout << "Enter choice (S)end, (R)ecieve, (N)umber, "
14              << "(L)ist, (D)ump, (C)lear, (Q)uit: ";
15         cin >> choice;
16         switch (choice) {
17             case 's': case 'S':
18                 cout << "Enter queue name and message: ";
19                 cin >> qname;
20                 cin.getline(message, sizeof(message));
21                 if ( (mq=mqlist.GetMessageQueue(qname)) != NULL)
22                     mq->Send(message);
23                 break;
24
25             case 'r': case 'R':
26                 cout << "Enter queue name: ";
27                 cin >> qname;
28                 if ( (mq=mqlist.GetMessageQueue(qname)) != NULL) {
29                     if (mq->Receive(message) == SUCCESS)
30                         cout << "Received message from queue "
31                              << qname << ": " << message << endl;
32                     else
33                         cout << "No message from from queue "
34                              << qname << endl;
35                 }
36                 break;
37
38             case 'l': case 'L':
39                 int i;
40                 for (i=0; i < MAX_QUEUES; i++) {
41                     if ( (mq=mqlist.GetMessageQueue(i)) != NULL)
42                         cout << "Name: " << mq->name << " count: "
43                              << mq->Count() << endl;

```

Continued

```

44         )
45         break;
46
47     case 'd': case 'D':
48         cout << "Enter queue name: ";
49         cin >> qname;
50         if ( (mq=qlist.GetMessageQueue(qname)) != NULL)
51             mq->Dump();
52         break;
53
54     case 'n': case 'N':
55         cout << "Enter queue name: ";
56         cin >> qname;
57         if ( (mq=qlist.GetMessageQueue(qname)) != NULL)
58             cout << "Size of queue " <<
59                 qname << ": " << mq->Count() << endl;
60         break;
61
62     case 'c': case 'C':
63         cout << "Enter queue name: ";
64         cin >> qname;
65         if ( (mq=qlist.GetMessageQueue(qname)) != NULL)
66             mq->Clear();
67         break;
68
69     case 'q': case 'Q':
70         return 0;
71     }
72 }
73 SOM_DestroyLocalEnvironment(__SOMEnv);
74
75     return(0);
76 }

```

Makefile (mqueue/makefile):

```

1  all: mqueue.idl mqueue.dll tstmq.exe
2
3  ICCOPTS = -DEXPORT:_Export /I ..\include
4
5  mqueue.dll: mqueue.hh mqueue.cpp
6      icc $(ICCOPTS) /Ti+ /Ge- /B"/NOE" \
7          mqueue.cpp mqueue.def ..\bin\sommem1.obj
8      implib mqueue.lib mqueue.dll
9
10 tstmq.exe: mqueue.hh mqmgr.h tstmq.cpp mqmgr.h mqmgr.cpp
11     icc $(ICCOPTS) /Ti+ /B"/NOE" tstmq.cpp mqmgr.cpp \
12         ..\bin\sommem1.obj mqueue.lib
13
14 mqueue.idl: mqueue.hh
15     icc $(ICCOPTS) mqueue.hh
16     sc -sir -u -p mqueue.idl
17     sc -sdef mqueue.idl

```


Programming Considerations and Common Problems

Problems Saving Data

If your data is not being saved, or if the datastore looks like binary even though you specified ASCII, there are a number of possible causes for this problem. First, make sure that all the attributes you want to be saved are specified as persistent in the IDL. Ensure that the class has been registered in the interface repository correctly, and that the **SOMIR** path correctly references that interface repository as discussed in Chapter 7. If you have any persistent attributes that are private or protected data members, you must compile the IDL with the **-p** option of the SOM compiler. Next, make sure that all objects you want to save, particularly if they are in a hierarchy, have been assigned an object ID.

Determining What Is Actually Being Saved/Restored

It is helpful for debugging persistent SOM programs to override the **sompActivated** and **sompPassivate** methods and use them to trace the saving and restoring being done by the program.

No-Argument Constructor Missing

You may get the following error when running a program that attempts to restore a persistent object:

```
DTS C++ class error: missing no-argument constructor
called on object of class %1.
Error: Current method not defined on the target object.
"somcls.c": 655: SOM Error - code = 2-006-9, severity = Fatal.
```

This occurs when the class for an object that is being restored does not have a default constructor (that is, a constructor that takes no arguments). The Persistence Framework restores an object first by creating an object through **somNew** and then by calling the **_set** attribute method to assign the value to each restored attribute. **somNew** calls **somDefaultInit** for the class, which maps to the C++ default constructor, which is why you must supply this method.

Exceptions When Freeing Memory

If you are getting an exception when deleting storage, keep in mind that the storage may have been allocated by SOM and, if so, you must use **somFree**

to release that storage. With DTS C++, if the storage is for a SOM object, then **somMalloc** is used implicitly. But for data types such as strings, SOM will allocate the storage when the object is restored, so you must use **somFree** to restore it because the DTS C++ compiler uses its own memory management routines. The easiest way to deal with this situation is to overload the global **new** and **delete** operators to use **SOMFree** and **SOMMalloc**, as described in Chapter 8. That way you can continue to use **new** and **delete** for all your memory management, but you don't have to worry about which allocator to call.

Migration Considerations for POSSOM

If you are implementing a persistent class, and POSSOM is available for your platform, then you should use it. If it is not, then you can use PSOM, but keep in mind the following changes that you will need to make to your class as you implement it to enable future migration to POSSOM.

Class Implementation

From a class implementation perspective, the following are some general guidelines covering the changes you will need to make to migrate a class from PSOM to POSSOM:

Changes:

- ♦ Change the class so that it inherits from **somStream::Streamable** and a POSSOM class, such as **somPersistencePO::PO**, instead of **SOMPPersistentObject**.

Removals:

- ♦ Any **persistent** attribute modifiers—they are not necessary for POSSOM.
- ♦ Overrides of **sompIsDirty**, **sompPassivate**, and **sompActivated**.

Additions:

- ♦ **init_for_object_creation**, **init_for_object_reactivation**, **init_for_object_copy**, **uninit_for_object_more**, **uninit_for_object_passivation**, and **uninit_for_object_destruction**
- ♦ **externalize_to_stream** and **internalize_to_stream**

Class Client

From a client perspective, the major difference between PSOM and POSSOM is in how persistent IDs are created and assigned and how objects are manipulated through the persistent datastore. With PSOM, objects are

saved, restored, and deleted through invoking methods against a Persistent Storage Manager, passing the persistent object as a parameter. With POSSOM, the persistent datastore is manipulated by invoking methods directly against the persistent object itself.

If you are writing an application that requires persistence and you will be using PSOM because POSSOM is not yet available for your platform, be aware of the differences in order to make eventual porting to POSSOM as simple as possible. Migrating the class itself to POSSOM is not that difficult, but you should try to isolate PSOM-specific client code as much as possible, as this is where most of the changes will likely be necessary.

The following example shows one possibility for defining a class to be more easily migrated. I defined two new methods for the class: `store` and `restore`, corresponding to the POSSOM methods. Clients can invoke these methods directly against the object to perform the save/restore operations.

In the class implementation code for the `store` and `restore` methods, care must be taken so that two objects do not have the same ID assigned to them. PSOM keeps track of this information, and will generate an error when this occurs. Because `sompRestoreObject` returns a newly created object, the data must be transferred to the current object, and the new object deleted, as shown at lines 23 through 25.

In the client program, I isolated the code to assign persistent IDs and delete a persistent object to the functions `assignPersistentID` and `deleteObject`. These functions isolate the PSOM-specific code so that the client can be more easily migrated to POSSOM. Because the `sompDeleteObject` method also deletes the in-memory copy of the object, I could not make this a method of the `PCount` class.

Definition of POSSOM-Enabled PCount Class (possompcount.hh):

```

1  #include <somp.hh>
2
3  class PCount : public SOMPPersistentObject {
4      #pragma SOMClassName(*, "PCount")
5      #pragma SOMIDLPass(*, "Implementation-End", \
6          *dllname = \"pcount.dll\";*)
7      boolean sompIsDirty();
8      boolean isIdSet;
9      #pragma SOMIDLTypes(*, SOMPPersistentStorageMgr)
10 protected:
11     void copyMembers(PCount &);
12 public:
13     PCount() { count = 0; isIdSet = FALSE; }
14
15     // possom methods
16     void store(void * = NULL);
17     void restore(void * = NULL);
18
19     static SOMPPersistentStorageMgr psom;
20     // persistent id

```

Continued

```

21     SOMPPersistentId *p;
22     #pragma SOMIDLTypes(*, SOMPPersistentStorageMgr)
23     #pragma SOMAttribute(p, noset)
24
25     int count;
26     #pragma SOMAttribute(count, noset)
27     #pragma SOMIDLPass(*, "Implementation-Begin", \
28         "count: persistent;")
29 };

```

Implementation of POSSOM-Enabled PCount Class (possom/pcount.cpp):

```

1  #include <iostream.h>
2  #include "pcount.hh"
3
4  SOMPPersistentStorageMgr PCount::psm;
5
6  void PCount::store(void *)
7  {
8      // can only have one object with given id
9      if (! isIdSet) {
10         sompInitGivenId(p);
11         isIdSet = TRUE;
12     }
13     psm.sompStoreObject(this);
14 }
15
16 void PCount::restore(void *)
17 {
18     if (! psm.sompObjectExists(p))
19         return;
20     cout << "restoring" << endl;
21     PCount *tmp =
22         (PCount *)psm.sompRestoreObject(p);
23     copyMembers(*tmp);
24     delete tmp;
25     sompInitGivenId(p);
26     isIdSet = TRUE;
27 }
28
29 void PCount::copyMembers(PCount &from)
30 {
31     count = from.count;
32 }
33
34 void PCount::_set_p(SOMPPersistentId *pid)
35 {
36     p = pid;
37 }
38
39 void PCount::_set_count(int newValue)
40 {
41     count = newValue;
42     // set the dirty bit for the object

```

```

43     this->sompSetDirty();
44 }
45
46 boolean PCount::sompIsDirty()
47 {
48     // return the dirty bit for the object
49     return this->sompGetDirty();
50 }
51
52 SOMEXTERN void SOMLINK EXPORT SOMInitModule(
53     long major, long minor, string className)
54 {
55     PCountNewClass(major, minor);
56 }

```

Client of POSSOM-Enabled PCount Class (possom/tstpcnt.cpp):

```

1  #include <iostream.h>
2  #include "pcount.hh"
3  #include "check.h"
4
5  void deletePObject(PCount *&pobj);
6  void assignPersistentId(PCount *pobj);
7
8  int main(int argc, char *argv[])
9  {
10     __SOMEnv = SOM_CreateLocalEnvironment();
11
12     // instantiate the persistent object
13     PCount *pobj = new PCount;
14     assignPersistentID(pobj);
15
16     pobj->restore();
17     checkError(__SOMEnv);
18     ++pobj->count;
19     cout << "Count is " << pobj->count << endl;
20
21     if (pobj->count < 3) {
22         // save object through persistent storage manager
23         pobj->store();
24     } else {
25         deletePObject(pobj);
26     }
27
28     // free the object
29     if (pobj)
30         delete pobj;
31
32     SOM_DestroyLocalEnvironment(__SOMEnv);
33 }
34
35
36 void assignPersistentId(PCount *pobj)
37 {

```

Continued

```
38     SOMPPersistentId *pid = new SOMPPersistentId;
39
40     // set the file name,
41     // format defaults to ASCII, offset to 0
42     pid->sompSetIOGroupName("pfile.dat");
43
44     pobj->p = pid;
45 }
46
47 void deletePObject(PCount *&pobj)
48 {
49     pobj->psm.sompDeleteObject(pobj->p);
50     pobj = NULL;
51 };
```

References

- [ANSI, 1994] ANSI COBOL-97 Working Paper, December 20, 1994.
- [Danforth, 1994] Danforth, S., P. Koenen and B. Tate. *Objects for OS/2*. New York: John Wiley & Sons, 1994.
- [Danforth, 1995] Danforth, S. and J. Rogers. "Mapping DirectToSOM C++ to SOM IDL." IBM Document, 1995.
- [Ellis, 1990] Ellis, M. and B. Stroustrup. *The Annotated C++ Reference Manual*. Reading, MA: Addison-Wesley, 1990.
- [Hamilton, 1995] Hamilton, J., R. Klarer, M. Mendell, and B. Thomson, "Using SOM with C++," *C++ Report*, July 1995.
- [Hamilton, 1996a] Hamilton, J. "A Model for Implementing an Object-Oriented Design without Language Extensions," *SIGPLAN Notices*, January 1996.
- [Hamilton, 1996b] Hamilton, J. "Reusing Binary Objects with C++," *C++ Report*, March 1996.
- [Hamilton, 1996c] Hamilton, J. "Interlanguage Object Sharing with SOM," Proceedings of the 2nd USENIX Conference on Object-Oriented Technologies and Systems, June 1996.
- [IBM, 1994] *IBM VisualAge SOMSupport Guide*, 1994. (Online document supplied with IBM VisualAge for Smalltalk for OS/2 version 2.0.)
- [IBM, 1995a] *IBM VisualAge for C++ for OS/2 Version 3.0 Programming Guide*, IBM Document S25H-6958.
- [IBM, 1995b] *IBM VisualAge for COBOL for OS/2 Version 1.1 Programming Guide*, IBM Document SC26-8423.

- [Lau, 1994] Lau, C. *Object-Oriented Programming, Using SOM and DSOM*, New York: John Wiley & Sons, 1994.
- [Pennello, 1995] Pennello, T. "A DirectToSOM C++ Cookbook," MetaWare Incorporated, 1995.
- [Sessions, 1996a] Sessions, R. *ObjectWatch on SOM*, February 1996 (<http://www.fc.net/~roger/owatch.htm>)
- [Sessions, 1996b] Sessions, R. *ObjectWatch on SOM*, March 1996 (<http://www.fc.net/~roger/owatch.htm>)
- [Stroustrup, 1991] Stroustrup, B. *The C++ Programming Language*, 2nd Edition. Reading, MA: Addison-Wesley, 1991.

- `_ClassObject`, 119, 365
- `_Optlink linkage`, 131
- `_System linkage`, 131
- `l`, with `SOMReleaseOrder` pragma, 79
- `/B/NOE`, 235
- `/Fr`, 44, 79–81, 99
- `/Fs`, 80, 81, 155, 156
- `/Ga`, 16, 80–82
- `/Gb`, 80, 82, 220
- `/Gd`, 238
- `/Gz`, 80, 82, 139
- `/qautoimport`, 111
- `/Tdp`, 97, 127
- `/Tm`, 100
- `/Xs`, 80, 82
- `<class>CClassData`, 20, 70, 82, 105, 125, 127, 140, 142, 387
- `<class>ClassData`, 20, 35, 37, 50, 70, 77–79, 82, 105, 120, 125, 127, 140–142, 144, 147, 246, 247, 387, 409
- `<class>NewClass`, 20, 70, 105, 111, 113, 125–127, 140, 280
- `<som.hh>`, 11, 15, 24, 54, 81, 159, 398
- `<somapi.h>`, 140, 377
- `<somcls.hh>`, 24, 404
- `<somcorba.h>`, 93, 393
- `<somdd.hh>`, 204
- `<somh.h>`, 398
- `<somnew.hh>`, 100, 104
- `<somobj.hh>`, 24, 152, 159, 399
- `<mosos.h>`, 307
- `__isDTSClass`, 120, 121
- `__PRIVATE__`, 164, 165
- and `SOM compiler -p` option, 165, 439
- `__PROTECTED__`, 164, 165
- and `SOM compiler -p` option, 165, 439
- `__SOM_ENABLED__`, 80
- `__SOMEnv`, 27, 93–96, 129, 142, 204, 248, 268, 325, 358
- `__declspec`, 26, 106, 108
- `__DLL_InitTerm`, 112
- `__Export`, 106, 108
- `__get__set` attribute methods, 54–66, 76, 122, 181, 220, 283
- See Also `SOMAttribute` pragma
- `__Import`, 106, 108
- 10 special `SOMObject` methods, 22, 79, 95, 122, 148, 169, 294
- Activation, of persistent object, 308, 309
- `alloca`, 100, 135, 138
- `already_streamed` method, 340
- `Arrays`, 45, 85, 90, 139, 175, 241, 242, 274
- and passing parameters with `Smalltalk`, 288
- `Attribute`, see `SOMAttribute` pragma
- `Backing data`, 55, 56, 63, 181, 217, 220, 283
- `bind` method, 312, 313, 316, 317
- `bind_context` method, 316
- `bind_new_context` method, 316
- `caller-owned memory management`, 221–225, 230, 267, 272
- See Also `DSOM`, memory management
- `Callstyle`, 66, 95, 142, 149, 161, 284
- See Also `SOMCallStyle` pragma
- `capture` method, 353
- `Case-sensitivity`, `SOM` names, 68, 70, 75
- `Casting`, pointers to `SOM` objects, 92, 129, 138, 212
- `checkError` function, 205
- `Class object`, 5, 23, 24, 74, 101, 116, 119, 144, 210, 277, 280
- creating, 9, 11, 37, 43, 66, 69, 82, 111–115, 134, 139
- `Concurrency Service`, 306
- `constant_random_id` method, 330, 340
- `control structure`:
 - assignment, destruction, and initialization, 146, 148–150, 161, 294
- `CORBA`, 4, 7, 8, 66, 157, 173, 175, 192, 202, 221, 247, 294, 393
- attribute, see `SOMAttribute` pragma
- `CosNaming::NamingContext`, 313–317
- `bind` method, 312, 313, 316, 317
- `bind_context` method, 316
- `bind_new_context` method, 316
- `new_context` method, 316
- `unbind` method, 313, 317
- `CosStream::Streamable class`, 333, 340
- `CosStream::StreamIO class`, 334
- `externalize_to_stream` method, 333–335, 345, 359, 440
- `internalize_from_stream` method, 333–335, 345, 359, 440
- `read_object` method, 340, 359
- `write_object` method, 339, 359
- `write_object_value` method, 339, 359
- `write_short` method, 358
- `write_string` method, 358
- `CPPFILTER` utility, 127, 130, 153
- `Data token`, 123, 124, 142, 147
- `deactivate_impl` method, 250
- `Debugger`, 129, 264, 265
- `Default constructor`, 21, 119, 138, 149, 245, 283, 433, 439
- `delete operator`, see `DSOM`, memory management
- `Direct data access`, 76, 82, 217–220
- See Also `DSOM`, accessing data members
- `DirectToSOM C++`:
 - client bindings, 195
 - compiler options:
 - `/B/NOE`, 235
 - `/Fr`, 44, 79–81, 99
 - `/Fs`, 80, 81, 155, 156
 - `/Ga`, 16, 80–82
 - `/Gb`, 80, 82, 220
 - `/Gd`, 238
 - `/Gz`, 80, 82, 139
 - `/qautoimport`, 111
 - `/Tdp`, 97, 127
 - `/Tm`, 100
 - `/Xs`, 80, 82
 - Header Files, 119
 - Inline functions, 85
- `DirectToSOM C++ class`:
 - as concrete class, 50
 - forward declaration of, 121, 122
 - inheritance, 83, 84
 - instance data, 38, 146–148
- `DLL`, see `SOM DLL`
- `dllimport`, 106, 108
- `dllname`, 112, 190
- `DSOM`, 203, 174
 - accessing data members, 217–220
 - and default constructor, 245, 246
 - and standard I/O, 247
 - common problems, 264–269
 - creating remote objects with a factory, 210–214
 - daemon, `somdd`, 210, 249, 266
 - data access checklist, 220
 - demarshalling, 210
 - direct data access, 219, 220
 - `DSOM 2.x`, 269–274
 - environment setup, 207–210
 - error handling, 247, 248
 - error log, 265
 - factory service, 211

DSOM (Continued)

- finding existing objects, 214–217
- marshaling information, 207
- marshaling, 210
- memory management, 221–240
 - client, 225–230
 - caller-owned, 221–225, 230, 267, 272
 - dual-owned, 221, 225, 231, 272
 - mapping C++ memory routines to SOM, 235–237
 - mapping SOM memory allocation to C++, 237–240
 - object-owned, 223–225, 230–232
 - object_owns_results, 232
 - ORBFREE vs. SOMFREE vs. delete, 234
 - server, 230–233
 - suppress_inout_free, 223, 227, 234, 272
- Object Request Broker (ORB)
 - class, 211
- parameter passing with, 240–245
 - arrays, 241, 242
 - parameter directional attributes, 240
- SOMFOREIGN types, 242–245
- server process, 204, 210
- server program, customizing, 238, 248–252
- shallow vs deep copy, 233
- som_cfg utility, 208, 269, 317
- static members with, 246, 247
- stringified proxy, 214
- stub DLL, 247
- workgroup, 203
- workstation, 203, 210
- See Also Implementation Repository; Interface Repository
- dual-owned memory management, 221, 225, 231, 272
- dual_owns_parameters, 232
- duplicate method, 254
- Dynamic marshaling
 - see DSOM, parameter passing, SOMFOREIGN types
- Embedded SOM object, 88–90, 184–186
- Emitter Framework, 7, 13
- Environment variables:
 - INCLUDE, 25
 - SOMBASE, 24, 110, 195, 208
 - SOMIR, 202, 208
 - SOMP_PERSIST, 420
 - TEMPINC, 97
- Error handling, 66, 93–96, 124–131, 247, 264–269
- Event Management Framework, 7
- Event Service, 306

execute_next_request method, 250

- Explicit SOM mode, 16, 54, 81
- Exported symbols, see SOM class data structures
- ExtendedNaming::ExtendedNaming
 - Context, 211, 312–320
 - find_any method, 211–212
- Externalization Service, 332–340
- externalize_to_stream method, 333–335, 345, 359, 440
- Factory Service, 210–214
- FactoryFinder class, 324–326, 334
- find_any method, 211–212
- find_impldef, 249
- find_impldef_by_alias, 249
- garbage collection:
 - with PSOM, 429
 - with Smalltalk, 277
- IDL, 155–202
 - context expression, 192
 - cxdecl modifier, 160
 - cxmap modifier, 175, 176
 - enumerations, 175, 176
 - generating an IDL file, 155–157
 - generating DirectToSOM C++ client bindings, 195–198
 - implementation statement, 157
 - interface declaration, 157
 - modifier statements, 158
 - module statement, 197
 - porting to DirectToSOM C++, 200, 201
 - raises expression, 102
 - string type, 175
- IDL generation, 155–202
 - appending modifiers, 192
 - of attributes, 181–183
 - of constructors, 186–188
 - dts_fwd, 177
 - of embedded objects, 184–186
 - generating IDL-specific information from C++, 190–194
 - mapping by file, 179, 180
 - mapping types from C++ to IDL, 174–179
 - modifying generated declarations, 191
 - nested classes, 177
 - of operators, 173
 - parameter direction (in/out/inout), 173
 - of sequences, 192–194
 - of templates, 188–190
 - of SOMFOREIGN type, 176, 177
 - som3InitCtrl, 186
 - somInitCtrl, 186

IDL modifiers:

- align, 160
- declarationorder, 188–190
- directInitClasses, 164
- dllname, 112, 190
- dual_owns_parameters, 232
- impctx, 177
- init, 186
- memory_management=corba, 221
- nested, 177
- nests, 177
- noenv, 169
- nonstatic, 160
- nonstaticaccessors, 160, 181
- noself, 169
- offset, 160
- object-owned, 223–225, 230–232
- object_owns_results, 232
- override, 160, 169
- private, 165
- privateaccessors, 181
- procedure, 169
- protected, 165
- protectedaccessors, 181
- public, 160, 165
- publicaccessors, 181
- reintroduce, 169
- releaseorder, 160, 181
- size, 160
- suppress_inout_free, 223, 227, 234, 272
- imod, SOM compiler emitter, 111, 266
- impl_is_ready method, 249
- implementation Repository, 207, 209, 249, 264, 310, 326
- regimpl utility, 209, 212, 250, 264, 311
- implicit SOM mode, 16, 54, 81
- init_for_object_activation method, 309, 345
- init_for_object_copy method, 309
- init_for_object_creation method, 309, 325, 333–335, 341, 345, 347, 358, 440
- inline functions, 85
- instance data, 38, 146–148
- Interface Definition Language, see IDL
- interface repository, 7, 112, 155, 201, 207, 221, 249, 266, 275–279, 439
- IRDUMP command, 202, 265–167
- interlanguage object sharing, 275–304
- examples, 284–304
- OO COBOL, 278–282
 - inheriting from a C++ class, 302
- Smalltalk, 276–278
- asINOUTParameter, 288
- asOUTParameter, 288
- garbage collection, 277
- style guidelines, 282–284

- internalize_from_stream method, 333–335, 345, 359, 440
- introducing class, 18, 37, 66, 79, 95, 144
- IRDUMP command, 202, 265–267
- is_identical method, 330, 340
- IsValidRemoteObject function, 214
- language bindings, 6–10, 12, 13, 53, 100, 195
- Life Cycle Service, 305, 324–329
- make_persistent_ref method, 341
- memory management, 100–105, 116
 - overloading new and delete, 100–102
 - providing class-specific new and delete, 102, 103
- SOMMalloc and SOMFree, 104, 105
 - See Also DSOM, memory management
- Message queue example:
 - original, 26–34
 - with DSOM, 252–264
 - with Naming Service, 319–324
 - with POSSOM, 357–376
 - with PSOM, 429–438
 - with RRBC, 44–50
- Metaclass, 7, 23, 120, 135, 279, 283
- Metadata, 310, 353
- Method promotion, 39
 - See Also RRBC
- Method resolution, 9, 129, 139, 141, 247
- Method table, 5, 37, 129, 135, 142–147
 - See Also parent method table
- Method tokens, 140, 144–146
- Module, SOM, 75, 195, 197–199, 306
- Multi-thread capable, 335, 348
- Multiple inheritance, 146, 279
- Name mangling, 22, 75–77, 151–153, 162–164, 282
 - See Also SOMClassName;
 - SOMDataName;
 - SOMMethodName;
 - SOMNoMangling
- naming context, 211, 312–320, 326, 335, 341
- Naming Service, 208, 211, 214, 267, 312–324
- new operator, see Memory management
- new_context method, 316
- nodata, SOMAttribute pragma keyword, 63–65, 181, 246
- noget, SOMAttribute pragma keyword, 56–58, 65, 130, 246
- noset, SOMAttribute pragma keyword, 56–58, 65, 130, 246, 427
- Object Identity Service, 329–332
- object-owned memory management, 223–225, 230–232
- object_owns_results memory management, 232
- object_to_string method, 214
- offsetof, 85–87
- Object IDL (OIDL), see Callstyle
- OMG, see CORBA
- OO COBOL, see Interlanguage object sharing
- Opaque marshaling, see DSOM, parameter passing, SOMFOR-EIGN types
- operator=, 23, 79, 148–151, 173, 284
- ORB class:
 - object_to_string method, 214
 - resolve_initial_references method, 211, 312
 - string_to_object method, 214
- ORBFree, 234
 - See Also DSOM, memory management
- Parent method table, 142, 146
- passivate_all_objects method, 353, 356
- passivation, 308, 347, 353, 356, 358
- PDS, see Persistent Data Service
- Performance, run-time, 50, 56, 123, 220, 283
- Persistence, 135, 307, 310
 - Persistent Object Service (POS-SOM), 341–376
 - Persistence SOM (PSOM), 415–444
 - persistence identifier (PID):
 - POSSOM, 346–348, 353, 358–360
 - PSOM, 417–423
 - persistent reference, 308, 341–345, 353, 359
- Persistent Data Service (PDS), 346, 348
- Persistent Object Manager (POM), POSSOM, 346, 348
- Persistent Storage Manager, PSOM, 416, 418
- Pointers to members, 17, 122
- POM, see Persistent Object Manager
- Pragmas, DirectToSOM C++:
 - conventions used with, 52
 - priority, 237
- SOM, 53
- SOMASDefault, 15, 53, 81
- SOMAttribute, 54–66, 74, 139, 181, 217, 232, 283, 288
 - See Also SOMAttribute pragma
- SOMCallStyle, 66
 - See Also Callstyle; Object IDL
- SOMClassInit, 66
- SOMClassName, 53, 68, 77, 99, 115, 152, 162, 282
 - with SOM module, 195
- SOMClassVersion, 69
- SOMDataName, 70, 76, 160, 162, 282
- SOMDefine, 20, 70–72, 99, 125
- SOMIDLDecl pragma, 174, 191–193, 227, 240, 243, 267
- SOMIDLPass pragma, 112, 190, 193, 221
- SOMIDTypes pragma, 74
- SOMMetaClass pragma, 74
- SOMMethodAppend pragma, 52, 192
- SOMMethodName pragma, 75, 152, 162, 173, 192, 240, 282
- SOMModule pragma, 75, 197, 307
- SOMNoDataDirect pragma, 76, 82, 220, 269
- SOMNoMangling pragma, 70, 76, 152, 162, 173, 192, 282
- SOMNonDTS pragma, 77, 195, 200 summary, 52
- priority pragma, 237
- Private base class, 43, 122, 165
- privatedata, SOMAttribute keyword, 56
- protecteddata, SOMAttribute keyword, 56, 181
- Proxy, 203–205, 210–212, 214, 225, 267, 307–309
- publicdata, SOMAttribute keyword, 56, 181, 220
- read_object method, 340, 359
- readonly, SOMAttribute keyword, 56, 62, 78, 181, 253
- regimp utility, 209, 212, 250, 264, 311
 - See Also Implementation Repository
- reinit method, 353
- release method, 212, 307
- release order, 5, 18, 35–44, 77–79, 141, 148, 160, 165, 170
- Release-to-release binary compatibility (RRBC), 35–50, 77–80
 - supported changes, 39–43
 - unsupported changes, 43–44
- Replication SOM (RSOM), 7
- resolve_initial_references method, 211, 312
- RRBC, see Release-to-release binary compatibility
- sc, SOM compiler command, 24, 28, 110, 157, 195, 202
 - See Also SOM compiler
- Security Service, 306
- server alias, 209, 212, 249, 264, 268

- server implementation, 209, 212, 249, 271, 310–312, 335, 342, 348, 353
- sizeof, 85, 104, 143
- Smalltalk, 105, 115, 276–278, 285–302
 - See Also Interlanguage object sharing
- SOM:
 - exceptions, see Error handling
 - class data structures, 20, 39, 82, 125–127, 145–148, 152, 197, 247, 280
 - class manager, see SOMClassMgr class
 - class name, 68, 77, 99, 115, 151–153, 163, 197
 - methods kinds, 169
 - method name, 76–78, 152
- SOM compiler, 6–8, 13, 75, 157, 161, 195, 200–202, 275, 439
- imod emitter 111, 112, 266
- p option, 165, 431
- SOM DLL:
 - _DLL_InitTerm, 112
 - _Export, 106, 108
 - _Import, 106, 108
 - dllimport, 106, 108
 - dynamically loading, 116–119
 - enabling for dynamic loading:
 - SOM 3.0, 111–113
 - SOM 2.x, 113–115
 - exporting static symbols, 106–111
 - See Also DSOM, stub DLL
- SOM Object Adapter (SOMOA), 249, 310
- som_cfg utility, 208, 269, 317
- SOM_CreateLocalEnvironment function, 27, 96
- SOM_DestroyLocalEnvironment function, 96
- SOM_InitEnvironment function, 96, 248
- SOMAsDefault pragma, 15, 53, 81
- somAssign method, 151, 195, 196
- SOMAttribute pragma, 54–66, 74, 139, 181, 217, 232, 283, 288
 - nodata, 63–65, 181, 246
 - noget, 56–58, 65, 130, 246
 - noset, 56–58, 65, 130, 246, 427
 - privatedata, 56
 - protecteddata, 56, 181
 - publicdata, 56, 181, 220
 - readonly, 56, 62, 78, 181, 253
 - virtualaccessors, 56, 65
- See Also /qsvolattr compiler option
- SOMBASE environment variable, 24, 110, 195, 208
- somBuildClass, 140, 152, 153
- SOMCalloc, see Memory management
- SOMCallStyle pragma, 66
 - See Also Callstyle; Object IDL
- SOMClass class, 24, 67, 95, 101, 120, 134, 279, 283, 325, 404
- somGetName method, 67
- somNew method, 24, 116, 135, 205, 212, 245, 280, 283, 424, 439
- somNewNoInit method, 137, 205, 212, 283, 325
- somRenew, 136
- somRenewNoInit, 137
- somRenewNoInitNoZero, 137
- somRenewNoZero, 137
- SOMClassInit pragma, 66
- SOMClassMgr class, 116, 135
- somLocateClassFile method, 116
- SOMClassMgrObject global variable, 135
- SOMClassName pragma, 53, 68, 77, 99, 115, 152, 162, 282
 - with SOM module, 195
- SOMClassVersion pragma, 69
- SOMD_ImplDefObject global variable, 249
- SOMD_ImplRepObject global variable, 249
- SOMD_Init function, 204, 248
- SOMD_NoORBFree function, 234, 248
- SOMD_ObjectMgr global variable, 270
- SOMD_ORBObject global variable, 211, 214, 310, 312
- SOMD_ServerObject global variable, 249, 310
- SOMD_SOMOAObject global variable, 249
- SOMDataName pragma, 70, 76, 160, 162, 282
- somCreate function, 204
- somdmd, DSOM daemon, 210, 249, 266
- somDeleteObj method, 271
- somDefaultAssign method, see SOMObject class, 10 special methods
- somDefaultConstAssign method, see SOMObject class, 10 special methods
- somDefaultConstCopyInit method, see SOMObject class, 10 special methods
- somDefaultConstVAssign method, see SOMObject class, 10 special methods
- somDefaultConstVCopyInit method, see SOMObject class, 10 special methods
- somDefaultCopyInit method, see SOMObject class, 10 special methods
- somDefaultInit method, 136, 280, 424, 439
 - See Also Memory management
- somDefaultVAssign method, see SOMObject class, 10 special methods
- SOMDefine pragma, 20, 70–72, 99, 125
- somDestroyObject method, 270
- somDestruct
 - see SOMObject class, 10 special methods
- somdFindAnyServerByClass method, 271
- somdFindServerByName method, 271
- somdFindServersByClass method, 271
- somdNewObject method, 269
- SOMDObject class, 212, 223, 254, 307
 - duplicate method, 254
 - release method, 212, 307
- SOMDObjectMgr class, 270
- somDestroyObject method, 270
- somdFindAnyServerByClass method, 271
- somdFindServerByName method, 271
- somdFindServersByClass method, 271
- somdNewObject method, 269
- somdRefFromSOMObj method, 250, 310
- SOMDServer class, 249, 271, 310
- somdDeleteObj method, 271
- somdRefFromSOMObj method, 250, 310
- somdSOMObjFromRef method, 250, 310
- somdSOMObjFromRef method, 250, 310
- somdsvr program, 210, 248, 264, 310
- somenv.ini configuration file, 208, 265
- somExceptionFree function, 94, 248
- somExceptionID function, 93
- somExternalization::OSStream class, 333–335
- somExternalization::Stream class, 333
- somExternalization::StreamFactory, 333
- SOMFOREIGN type, 174–177, 184, 188, 227, 242–244
- somFree method, 205, 280, 307
- SOMFree function, see Memory Management

- somGetGlobalEnvironment function, 288
- somGetName method, 67
- somId type, 116
- somIdFromStr function, 116
- SOMIDLDecl pragma, 174, 191–193, 227, 240, 243, 267
- SOMIDLPass pragma, 112, 190, 193, 221,
- SOMIDLTypes pragma, 74
- SOMIR environment variable, 202, 208
- SOMLINK macro, 8, 67, 113, 131
- somLocateClassFile method, 116
- SOMMAlloc function, *see* memory management
- SOMMetaClass pragma, 74
- SOMMethodAppend pragma, 52, 192
- SOMMethodName pragma, 75, 152, 162, 173, 192, 240, 282
- SOMModule pragma, 75, 197, 307
- somNew method, 24, 116, 135, 205, 212, 245, 280, 283, 424, 439
- somNewNoInit method, 137, 205, 212, 283, 325
- SOMNoDataDirect pragma, 76, 82, 220, 269
- SOMNoMangling pragma, 70, 76, 152, 162, 173, 192, 282
- SOMNonDTS pragma, 77, 195, 200
- SOMOA class, 249
 - deactivate_impl method, 250
 - execute_next_request method, 250
 - impl_is_ready method, 249
- SOMObject class, 154, 84, 90, 95, 99, 133, 137, 212, 279, 283, 325
 - 10 special methods, 22, 79, 95, 122, 148, 169, 294
 - as private base class, 122
 - somFree method, 205, 280, 307
- SOMObject Object Services, 305–376
 - managed objects, 307–310
 - object life cycle model, 306
 - object services server, 310–312
- somOS::Server class, 310, 353
 - make_persistent_ref method, 341
 - passivate_all_objects method, 353, 356
- somOS::ServiceBase class, 306, 309–311, 325, 329
 - capture method, 353
 - init_for_object_activation method, 309, 345
 - init_for_object_copy method, 309
 - init_for_object_creation method, 309, 325, 333–335, 341, 345, 347, 358, 440
 - is_identical method, 330, 340
 - reinit method, 353
 - uninit_for_object_destruction method, 309, 325, 333–335, 341, 345, 347, 358, 440
 - uninit_for_object_move method, 309
 - uninit_for_object_passivation method, 309, 345
- somOS::serviceBasePRef class, 341, 345
- somossrv program, 310
- SOMP_PERSIST environment variable, 420
- sompDeleteObject method, 426, 441
- somPersistencePO::PO class, 345, 353, 440
- SOMPIdAssigner class, 417
- sompInitGivenId method, 417, 421
- sompInitNearObject method, 417, 422, 430
- sompInitNextAvail method, 417, 418
- somplast.id, 420
- sompMarkForCompaction method, 429
- sompObjectExists method, 424
- SOMPPersistentId class, 421
- sompSetIOGroupName method, 421
 - somutSetIdString method, 421
- SOMPPersistentObject class, 416
- sompInitGivenId method, 417, 421
- sompInitNearObject method, 417, 422, 430
- sompInitNextAvail method, 417, 418
- sompMarkForCompaction method, 429
- SOMPPersistentStorageMgr class, 416, 418
- sompDeleteObject method, 426, 441
- sompObjectExists method, 424
- sompRestoreObject method, 416, 423, 429
- sompStoreObject, 416, 418, 429
- sompStoreObjectWithoutChildren, 429
- sompRestoreObject method, 416, 423, 429
- sompSetIOGroupName method, 421
- sompStoreObject, 416, 418, 429
- sompStoreObjectWithoutChildren, 429
- SOMRealloc function, *see* memory management
- SOMReleaseOrder pragma, 36, 39–42, 44, 77–79
 - See Also* release order
- somRenew method, 136
- somRenewNoInit method, 137
- somRenewNoInitNoZero method, 137
- somRenewNoZero method, 137
- somResolveByName function, 116, 212, 294
- somStream::MemoryStreamIO class, 334
- somStream::StandardStreamIO class, 334
- somStream::Streamable, 333, 345, 357, 440
- somStream::StreamIO class, 333, 340
 - already_streamed, 340
- somStream::StringStreamIO class, 334
- somutSetIdString method, 421
- srgarbl utility, 429
- Static member:
 - data, 21, 36, 39, 44, 54, 77, 128, 142, 246, 283
 - function, 123, 131, 142, 246
 - See Also* DSOM, static members with
- Static marshaling, *see* DSOM, parameter passing, SOMFOR-EIGN types
- storage allocation, *see* Memory management
- Streamable objects, 332
- streamIOs, 333
- string_to_object method, 214
- suppress_inout_free, 223, 227, 234, 272
- TEMPINC environment variable, 97
- Templates, 88, 96–99, 120, 127, 188
 - template-include file, 97
- Thunk, 37, 44, 50, 123, 142, 146
- Transaction Service, 306
- Transparent SOM mode, 16, 54, 81
- unbind method, 313, 317
- uninit_for_object_destruction method, 309, 325, 333–335, 341, 345, 347, 358, 440
- uninit_for_object_move method, 309
- uninit_for_object_passivation method, 309, 345
- Version, of SOM class, 43, 69
- Virtual base class, 10, 84, 122
- Virtual functions, 18, 56, 65, 78, 142, 169
- virtualaccessors, SOMAttribute pragma keyword, 56, 65
- volatile, 57, 122, 130
- write_object method, 339, 359
- write_object_value method, 339, 359
- write_short method, 358
- write_string method, 358

CUSTOMER NOTE: IF THIS BOOK IS ACCOMPANIED BY SOFTWARE, PLEASE READ THE FOLLOWING BEFORE OPENING THE PACKAGE.

This software contains files to help you utilize the models described in the accompanying book. By opening the package, you are agreeing to be bound by the following agreement:

This software product is protected by copyright and all rights are reserved by the author, John Wiley & Sons, Inc., or their licensors. You are licensed to use this software on a single computer. Copying the software to another medium or format for use on a single computer does not violate the U.S. Copyright Law. Copying the software for any other purpose is a violation of the U.S. Copyright Law.

This software product is sold as is without warranty of any kind, either express or implied, including but not limited to the implied warranty of merchantability and fitness for a particular purpose. Neither Wiley nor its dealers or distributors assumes any liability for any alleged or actual damages arising from the use of or the inability to use this software. (Some states do not allow the exclusion of implied warranties, so the exclusion may not apply to you.)

A total insider's guide to programming cross-platform applications using DirectToSOM C++

Drawing upon her experience and those of her fellow team members, Jennifer Hamilton covers all the bases. Using many ready-to-use programming examples that you can paste into your own programs, she tells you everything you need to know to successfully:

- Develop applications in DirectToSOM C++
- Share objects across different languages
- Use the CORBA Object Services
- Avoid common problems encountered using DirectToSOM C++
- Get the most out of the DirectToSOM C++ RRBC support

With DirectToSOM C++ programmers can, for the first time, make full use of the power of SOM without having to leave a C++ development environment. Written by a member of the DirectToSOM C++ development team, this book gets you up and running with all the know-how and skills you need to work entirely in a cutting-edge C++ development environment while taking advantage of the full range of advanced SOM object technology, including release-to-release binary compatibility (RRBC), distribution, and cross-language support.



On the disk you'll find:

- Tons of reusable source code complete with marked files
- Header files currently only available through OSD

Programming with DirectToSOM C++ is a must-have for all VisualAge C++ programmers and developers.

JENNIFER HAMILTON is part of IBM's C++ compiler development group and is responsible for the development of DirectToSOM C++ support. She is the author of two other computer books and more than 20 professional articles.

WILEY COMPUTER PUBLISHING

John Wiley & Sons, Inc.
Professional, Reference and Trade Group
605 Third Avenue, New York, N.Y. 10158-0012
New York • Chichester • Brisbane • Toronto • Singapore • Weinheim

Cover Design: Susan Zucker

ISBN 0-471-16004-0



9 780471 160045